Genetic Algorithms in Java Basics

Apress®

Java 遗传算法编程

[英] Lee Jacobson [美] Burak Kanber 著
王海鹏 译

- 来自Java专家的声音
- 用遗传算法解决类似旅行商的经典问题

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Genetic Algorithms
in Java Basics

Java
遗传算法编程

[英] Lee Jacobson　[美] Burak Kanber　著
王海鹏　译

人 民 邮 电 出 版 社
北　京

图书在版编目（CIP）数据

Java遗传算法编程 /（英）雅各布森（Lee Jacobson），（美）坎贝尔（Burak Kanber）著；王海鹏 译. -- 北京 : 人民邮电出版社，2016.12
ISBN 978-7-115-43731-0

Ⅰ. ①J… Ⅱ. ①雅… ②坎… ③王… Ⅲ. ①JAVA语言－遗传－算法－程序设计 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第260320号

版权声明

◆ 著 [英] Lee Jacobson [美] Burak Kanber
译 王海鹏
责任编辑 陈冀康
责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路 11 号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
三河市海波印务有限公司印刷

◆ 开本：800×1000 1/16
印张：13.25
字数：160 千字 2016 年 12 月第 1 版
印数：1－3 000 册 2016 年 12 月河北第 1 次印刷

著作权合同登记号 图字：01-2016-5918 号

定价：49.00 元

读者服务热线：(010)81055410 印装质量热线：(010)81055316
反盗版热线：(010)81055315

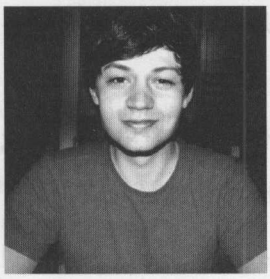
内容提要

当前，机器学习领域已经变得越来越流行，而遗传算法是机器学习的一个重要的子集。

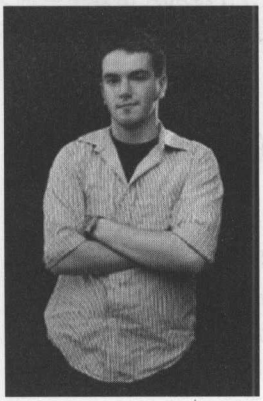
本书简单、直接地介绍了遗传算法，并且针对所讨论的示例问题，给出了 Java 代码的算法实现。全书分为 6 章。第 1 章简单介绍了人工智能和生物进化的知识背景，这也是遗传算法的历史知识背景。第 2 章给出了一个基本遗传算法的实现；第 4 章和第 5 章，分别针对机器人控制器、旅行商问题、排课问题展开分析和讨论，并给出了算法实现。在这些章的末尾，还给出了一些练习供读者深入学习和实践。第 6 章专门讨论了各种算法的优化问题。

本书适合机器学习爱好者阅读，尤其适合对遗传算法的理论和实现感兴趣的读者阅读参考。

作者简介



Lee Jacobson 来自英国布里斯托尔，是一名专业的自由职业软件开发者。他 15 岁时开始写代码，尝试编写游戏。他的兴趣很快转移到软件开发和计算机科学，并由此进入人工智能领域。在大学期间学习了遗传算法和其他优化技术后，他对这个主题产生了强烈的兴趣。他常常喜欢在晚自习时研究优化算法，如遗传算法，以及如何用它们解决各种问题。



Burak Kanber 是土生土长的纽约人，并进入了库伯高等科学艺术联盟学院（Cooper Union for the Advancement of Science and Art）学习。他拥有机械工程的学士学位和硕士学位，专注于控制系统、机器人技术、汽车工程和混合动力汽车系统工程。但是，软件一直是他终身的爱好，贯穿着 Burak 整个生命。

在库伯联盟学院时，Burak 开始为纽约市的初创公司提供咨询，在不同平台和不同行业中，帮助企业发展核心技术。置身于库伯联盟学院的艺术与设计氛围中，Burak 提升了产品设计的眼界和品位。

自 2009 年创立 Tidal Labs（一家技术公司，为企业影响者管理和内容营销开发获奖的软件）以来，Burak 一直在磨练自己在开发者运维、产品开发和机器学习等方面的技能。

Burak 喜欢晚上和未婚妻以及他们的猫 Luna 一起待在纽约的家里。

技术审阅者简介

Massimo Nardone 拥有意大利 Salerno 大学的计算机科学硕士学位。他曾作为一名 PCI QSA 及高级首席 IT 安全/云计算/ SCADA 架构师工作多年，目前在惠普芬兰任安全、云计算和 SCADA 首席 IT 架构师。他拥有超过 20 年的 IT 工作经验，其中包括安全、SCADA、云计算、IT 基础设施、移动、安全性和 WWW 技术领域，涉及国内和国际的项目。Massimo 曾担任项目经理、云计算 / SCADA 首席 IT 架构师、软件工程师、研发工程师、首席安全架构师和软件专家。他曾作为访问学者和导师，参与赫尔辛基大学的网络实验室（Aalto 大学）的工作。20 多年来，他一直在编程，并教授如何使用 Perl、PHP、Java、VB、Python、C / C ++和 MySQL 编程。他是 *Beginning PHP and MySQL*（Apress，2014）和 *Pro Android Games*（Apress，2015）的作者。他拥有 4 项国际专利（PKI、SIP、SAML 和 Proxy 领域）。

John Zukowski 目前是 TripAdivsor 的一名软件工程师，TripAdivsor 是全球最大的旅游网站（www.tripadvisor.com）。20 年来，他一直在使用 Java 技术，编写了 10 本 Java 相关的书籍。他的著作包括 Apress 出版的 *Java 6*、*Java Swing*、*Java Collections* 和 *JBuilder* 等相关技术图书，O'Reilly 出版的 *Java AWT* 图书和 Sybex 出版的 *Java* 入门书。他住在马萨诸塞州波士顿郊外，拥有约翰·霍普金斯大学的软件工程硕士学位。你可以关注他的 Twitter：http://twitter.com/javajohnz。

前　言

近年来，机器学习领域已经变得越来越流行。当然，这有很多原因，但处理能力的稳步增长，内存和存储空间的成本稳步下降，以及按需云计算的兴起，肯定起到了很大的作用。

但这些因素只是让机器学习的兴起成为可能，而无法解释其兴起的原因。是什么让机器学习如此引人注目？机器学习像一座冰山，顶端是一些创新和令人兴奋的领域，如计算机视觉、语音识别、生物信息学、医学研究，甚至是可以赢得 Jeopardy!游戏的机器（IBM 的沃森）。这些领域不应被低估，在未来几年中，它们绝对会成为巨大的市场驱动力。

然而，冰山水下的部分很大，也很成熟，足够今天使用，虽然我们很少看到年轻工程师声称“商务智能”是自己研究该领域的动机。机器学习（是的，今天的机器学习）让企业从复杂的客户行为中学习。机器学习帮助我们了解股市、天气模式、在拥挤的音乐会场地的群体行为，甚至可以预测下一次流感会在哪里爆发。

事实上，处理资源变得越来越便宜，很容易想象，在未来，机器学习在大多数企业的客户渠道、运营、生产和增长战略中起到核心作用。

然而，有一个问题。机器学习是一个复杂而困难的领域，放弃率很高；需要时间和精力来积累专业知识。我们正面临一个艰难而重要的任务：要让机器学习更容易，以跟上对该领域专家的不断增长的需求。到目前为止，我们已经落后于需求曲线。麦肯锡公司 2011 年的《大数据白皮书》预期：到 2018 年，对机器学习的人才需求将超

过人才供应的 50%～60%！虽然这让现有的机器学习专家在未来几年里日子很好过，但也阻碍了机器学习在不久的将来充分发挥影响的能力。

为何采用遗传算法

遗传算法是机器学习的子集。在实践中，遗传算法通常不是用来解决单一的、特定问题的最好算法。对任何一个问题，几乎总有更好的、更有针对性的解决方案！那么何必麻烦呢？遗传算法是一个极好的多用途工具，可以应用于许多不同类型的问题。这是瑞士军刀与合适的螺丝刀之间的差异。如果任务是拧紧 300 颗螺丝，你会跳起来找螺丝刀。但如果任务是拧几颗螺丝、割开一些布、在皮革上打一个孔，然后打开一瓶冰苏打水奖励自己的努力工作，那么瑞士军刀是更好的选择。

此外，我认为，遗传算法是整体研究机器学习的最佳入门。如果机器学习是一座冰山，遗传算法就是尖端的一部分。遗传算法有趣、令人兴奋且充满创新。遗传算法的模型基于自然生物过程，建立了计算世界和自然世界之间的连接。编写第一个遗传算法，观看从混乱和随机中出现的惊人结果，这让很多学生叹为观止。

机器学习冰山顶端的其他研究领域也同样令人兴奋，但它们往往关注的问题更狭窄，更难以理解。遗传算法则不然，它很容易理解，是有趣的实现，它们引入了所有机器学习技术都会使用的许多概念。

如果你对机器学习感兴趣，但不知道从哪里开始，就从遗传算法开始。你将学习一些重要概念，将来会带到其他领域中去，你会构建（不，你会获得）一个极好的多用途工具，可以用来解决许多类型的问题，并且不必学习高深的数学就能理解。

本书简介

这本书简单、直接地介绍了遗传算法。读者不需要在数学、数据结构和算法方面具备什么预备知识，就能理解本书的绝大部分内容，但是我们确实希望读者能适应中

级水平的计算机编程。虽然本书使用的编程语言是 Java，但我们没有使用任何特定的 Java 高级语言结构或第三方库。只要熟悉面向对象的编程，学习书中的例子就没有问题。在本书结束时，就能够舒服地用你选择的语言来实现遗传算法，不论它是面向对象语言、函数式语言还是过程式语言。

本书将引导你用遗传算法解决 4 个不同的问题。在这个过程中，你会学到一些技术，将来设计遗传算法时，你可以混合和适配这些技术。当然，遗传算法是一个庞大而成熟的领域，也有基础的严格数学形式，一本书不可能覆盖该领域的全部内容。因此，我们确定了一个原则：我们放弃了卖弄学问的讨论，避免了严格的数学形式，不涉及高级遗传算法。本书的目标是让你启动，快速运行实际的例子，让你有足够的基础，继续研究自己的高级主题。

源代码

本书呈现的代码是全面的，运行例子需要的一切都印在书中。但为了节省篇幅，我们通常省略了代码中的注释和 Java doc 注释块。请访问 http://www.apress.com/9781484203293，打开 Source Code/Downloads 选项卡，下载附带的 Eclipse 项目，其中包含了本书的所有示例代码。你会发现很多有益的注释和文档注释块，它们没有印在纸质书上。

通过阅读本书，运行它的例子，在最终成为机器学习专家的道路上，你就迈出了第一步。这可能改变你的职业生涯，但这取决于你。我们只能尽最大努力来讲授，提供一些工具，将来你需要用它们来构建自己的工具。祝你好运！

——Burak Kanber

目　录

第1章

简介

数字计算机和信息时代的崛起，已经彻底改变了现代的生活方式。数字计算机的发明，使我们能够让生活的许多领域数字化。这种数字化让我们将许多繁琐的日常任务外包给计算机，而这些任务以前可能需要人来完成。这方面的一个日常例子是字处理应用程序，它们内置拼写检查功能，可以自动检查文档的拼写和语法错误。

随着计算机变得越来越快，计算能力越来越强，我们已经能够用它们来执行越来越复杂的任务，如理解人类语言，甚至比较准确地预测天气。这种不断的创新，让我们能够将越来越多的任务外包给计算机。今天的计算常常能够每秒执行数十亿次操作，尽管它们在技术上能力很强，除非它们能够学习，让自己更好地适应提交给它们的问题，它们将永远只限于执行人类为它们写下的各种规则或代码。

人工智能及其子集遗传算法的领域，正开始解决今天的数字世界所面临的一些更复杂的问题。通过在真实世界应用程序中实现遗传算法，人们有可能解决较传统的计算方法几乎不能解决问题。

1.1 什么是人工智能

1950年，阿兰·图灵（数学家和早期计算机科学家）写了一篇著名的论文，题为“Computing Machinery and Intelligence（计算机器和智能）”，其中他提出问题：“计算机能思考吗”？他的问题导致了许多争论：智能到底是什么？计算机的根本局限是什么？

许多早期计算机科学家相信，计算机不仅能够展示类似智能的行为，而且通过短短几十年的发展，它们将达到人类的智力水平。这一观点由司马贺（Herbert A. Simon）在1965年提出，他宣称，“机器将有能力，在20年内，做任何人能做的工作”。当然，现在，在50年后的今天，我们知道司马贺的预测远离现实，但在当时，许多计算机科学家同意他的立场，并确立他们的目标是建立一个“强AI”机。一个强AI机就是一个机器，在面对给它的任何任务时，它至少像人类一样智能。

今天，在阿兰·图灵的著名问题被提出50多年之后，机器是否最终能够像人类一样思考，基本上仍然悬而未决。直到今天，他关于“思考”意味着什么的论文和思想，仍然引得哲学家和计算机科学家争论不休。

虽然我们还远远未能创造可以复制人类智能的机器，但在过去几十年里，我们无疑在人工智能上取得了显著的进步。自20世纪50年代以来，对“强AI”和发展媲美人类的人工智能的关注，已经开始转向对“弱 AI”的偏爱。弱AI是发展更狭窄领域的智能机，这在短期内更容易实现。这种较窄的关注点让计算机科学家创造实用的、貌似智能的系统，例如苹果公司的Siri和谷歌公司的自动驾驶汽车。

在创建弱 AI 系统时，研究人员通常会专注于建造一个系统或机器，它只拥有解决一个较小问题所需的“智能”。这意味着我们可以使用更简单的算法，使用更少的计算能力，同时还取得成果。相比较而言，强 AI 研究专注于建造一个智能机器，足以能够解决人类所能解决的任何问题。由于问题的范围很广，这使得构建强 AI 的最终产品的可能性小很多。

仅在几十年里，弱 AI 系统已经常见于我们的现代生活方式中。从下棋到帮助人类驾驶喷气式战斗机，弱 AI 系统已经证明，它们在解决问题时非常有用，这些问题一度被认为只有人类能解决。随着数字计算机变得越来越小、计算能力越来越强，这些系统的可用性很可能随时间推移而增加。

1.2 生物学类比

当早期的计算机科学家刚开始尝试建立人工智能系统时，他们常常向大自然寻求灵感，思考他们的算法的工作方式。通过创建模型，模仿自然界中发现的过程，计算机科学家能让他们的算法具有进化能力，甚至复制人类大脑的特征。通过实现仿生算法，这些早期开拓者第一次让机器有能力适应、学习和控制其环境的某些方面。

通过使用不同的生物学类比作为指导来开发人工智能系统，计算机科学家开创了不同的研究领域。当然，启发每个研究领域的不同生物学系统，都有自己的优势和应用。这本书中关注的一个成功的领域，就是进化计算，其中遗传算法构成了其主要的研究。其他领域专注于不大相同的方面，如人脑建模。这一领域的研究被称为人工神经网络，它利用了生物学神经系统的模型，来模拟其学习和处理数据的能力。

1.3 进化计算的历史

20 世纪 50 年代，进化计算首次作为一种优化工具被尝试，当时的计算机科学家将达尔文的生物进化论思想应用于候选解构成的种群。他们建立理论，认为有可能应用进化算子，如交叉（它是生物繁殖的模拟）和变异（这是新的遗传信息添加到基因组中的过程）。这些算子和选择压力共同作用，让遗传算法在一段时间后，能够“进化”出新的解。

在 20 世纪 60 年代，“进化策略”（应用自然选择和进化思想的一种优化技术）最初由 Rechenberg（1965，1973）提出，他的想法后来被 Schwefel（1975，1977）发展。其他计算机科学家当时在类似的研究领域独立地工作，如 Fogel L.J，Owens, A.J 以及 Walsh, M.J（1966 年），他第一个引入了进化编程的领域。他们的技术包括用有限状态机表示候选解，以及应用变异来创建新解。

在 20 世纪 50 年代和 60 年代，一些研究进化的生物学家开始用计算模拟进化。然而，是 Holland, J.H.（1975）在 20 世纪 60 年代和 70 年代首先提出并发展了遗传算法的概念。1975 年，在开创性著作《Adaption in Natural and Artificial Systems（自然与人工系统中的适应）》中，Holland 终于提出了他的想法。Holland 的书展示了达尔文的进化论可以如何被抽象并用计算机建模，用于优化策略。他的书解释了染色体如何建模为 1 和 0 的序列，通过实现自然选择中的技术，如变异、选择和交叉，拥有这些染色体的种群可以如何进化。

在 20 世纪 70 年代首次被引入后的几十年里，Holland 原创的遗传算法定义已经逐渐改变。这在某种程度上是因为，近期进化计算领域的研究人员偶尔将来自不同方法的思想融合在一起。虽然这模糊了许多方法学之间的界限，但

它为我们提供了丰富的工具，帮助我们更好地解决具体问题。在本书中，术语“遗传算法”既指 Holland 的遗传算法的经典定义，也指更宽泛的、今天的解释。

计算机科学家至今仍在研究生物学和生物系统，以便得到启发，创造更好的算法。最近的一个仿生优化算法是蚁群优化算法，它由 Marco, D. (1992)于 1992 年首先提出。蚁群优化算法对蚂蚁的行为建模，以此作为解决各种优化问题的方法，如旅行商问题。

1.4 进化计算的优势

智能机器在我们社会中的采用率正好确认了它们的有效性。我们用计算机来解决的绝大多数问题可以归结为相对简单的静态决策问题。随着可能的输入和输出的数量的增加，这些问题很快会变得更复杂，而且如果求解需要适应变化的问题，只会进一步复杂化。除此之外，一些问题也可能需要一种算法，搜索大量可能的解，试图找到一个可行解。根据需要搜索的解的数量，经典计算方法可能无法在允许的时间内找到可行的解，即便采用超级计算机也不行。正是在这种情况下，进化计算有了用武之地。

为了让你对传统计算方法能解决的典型问题有所了解，请考虑一个交通灯信号系统。这是相对简单的系统，只需要基本水平的智力操作。交通信号灯系统通常只有少量的输入，让它注意一些事件，如汽车或行人正等待通过路口。然后，它需要管理这些输入，并正确地改变信号灯，让汽车和行人有效地通过路口，而不会造成任何事故。尽管操作交通灯系统需要一定量的知识，但它的输入和输出都相当基本，可以让人来设计和编写一组指令来操作交通灯系统，这并没有太大的问题。

我们常常需要一个智能系统来处理更复杂的输入和输出。这可能意味着它不再简单，或许不可能让一个人编写一组指令，使机器将输入正确地映射到可行的输出。在这种情况下，如果问题的复杂性使得它不适合人类程序员编码解决，那么优化和学习算法就可以为我们提供一种方法，利用计算机的处理能力找到问题本身的解。这方面的一个例子是构建欺诈检测系统，该系统可以根据交易信息，识别欺诈交易。尽管交易数据和欺诈交易之间可能会出现关系，但它可能依赖于数据本身的许多精细模式。输入中的这些精细模式可能很难由人类来编码，这使它成为应用进化计算的很好的候选者。

如果人们不知道如何解决一个问题，进化算法也很有用。这方面的一个经典例子是 NASA（美国航空航天局），当时他们在寻找一个天线设计，满足对2006年的太空任务的所有要求。NASA 写了一个遗传算法，进化出一个天线设计，满足所有具体的设计约束，例如信号质量、尺寸、重量和成本。在这个例子中，NASA 不知道如何设计天线来满足所有的要求，所以他们决定写一个程序，而该程序可以进化出一个设计。

另一种我们也想采用进化计算策略的情况，就是问题是不断变化的，需要一个自适应的解。在建立算法预测股市时，就会遇到这样的问题。对这周股市预测准确的算法，可能无法在下周做出准确的预测。这是因为股市的模式和趋势不断变化，因此预测算法非常不可靠，除非在新模式出现时，它们能够快速适应不断变化的模式。进化计算可以帮助适应这些变化，提供一种方法，让预测算法具备必需的适应性。

最后，有些问题需要搜索大量的，或者也许是无限的可能解，来找到最好的或足够好的解。本质上，所有进化算法都可以看成是搜索算法，它们搜索一组可能的解，来寻找最好（或“最适合”）的解。如果将一个生物体的基因组中所有可能的基因组合看成是候选解，就可以看清这一点。生物进化非常擅长

于搜索这些可能的基因序列，找到足以适合其环境的解。在较大的搜索空间，很可能（即使用进化算法）找不到给定问题的最佳解。然而，这对大多数优化问题基本上不是问题，因为通常我们只需要足够好的解，把工作完成就可以了。

进化计算提供的方法，可以看成是一种“自下而上”的模式。这时候，从算法中涌现的所有复杂性来自简单的、基础的规则。另一种模式是“自上而下”的方法，需要在人编写的算法中展现所有的复杂性。遗传算法对于开发来说相当简单，这让它们成为有吸引力的选择，否则就需要复杂的算法来解决这个问题。

下面是一个问题特征列表，这类问题是采用进化算法的良好候选者：

- 如果问题足够困难，难以写代码来解决；
- 如果人不知道如何解决这个问题；
- 如果问题是不断变化的；
- 如果搜索每个可能解是不可行的；
- 如果可以接受“足够好”的解。

1.5 生物进化

生物通过自然选择的过程进化，这首先由达尔文（1859）在他的著作《物种起源》中提出。正是他的生物进化的概念，启发了早期的计算机科学家，转而用生物进化作为其优化技术的模型，并在进化计算的算法中实现。

因为在遗传算法中使用的许多思想和概念直接来自生物进化，对该主题有基本的了解，有利于深入了解这个领域。既然这样，在开始探索遗传算法之前，让我们先浏览一下生物进化的（有点简化）基础知识。

所有的生物都含有DNA，它编码了构成生物体的所有不同性状。DNA可以看成是生命的“说明书”，以便从头开始创建生物体。改变生物体的DNA会改变其性状，诸如眼睛和头发的颜色。DNA由单个基因组成，这些基因负责编码生物体的具体性状。

生物体的基因被分组放在染色体中，一套完整的染色体组成一个生物体的基因组。所有生物体至少具有一条染色体，但通常含有更多，例如人类有46条染色体，有些物种有超过1000条染色体。在遗传算法，我们通常将染色体称为候选解。这是因为遗传算法通常使用一条染色体来编码候选解。

对于特定性状的各种可能设置被称为“等位基因”，性状编码在染色体中的位置被称为“基因座”。我们将特定的基因组称为“基因型”，该基因型编码的物理生物体被称为“表现型”。

两个生物体交配时，来自两个生物体的DNA被带到一起，它们结合的方式导致产生的生物体（通常被称为的后代）从第一个亲代得到50%的DNA，并从第二个亲代得到另外50%的DNA。来自生物体DNA的基因偶尔会发生变异，为它提供双亲所没有的DNA。通过为种群增加以前没有的基因，这些变异为种群提供了遗传多样性。种群中的所有可能的遗传信息被称为种群的“基因库”。

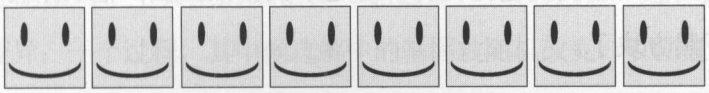
如果产生的生物体对环境适应得足够好，能够生存，它自己很可能交配，让它的DNA继续留在未来种群中。然而，如果产生的生物体对环境适应得不够好，不能生存并最终交配，它的遗传物质将不会传播到未来种群中。这就是为什么进化偶尔被称为适者生存，因为只有适者才能达到个体生存并传递它们的DNA。正是这种选择性的压力，慢慢将进化导向发现越来越适应、越来越好的个体。

生物进化的一个实例

为了阐明这个过程如何逐渐导致进化出越来越适应的个体，请考虑下面

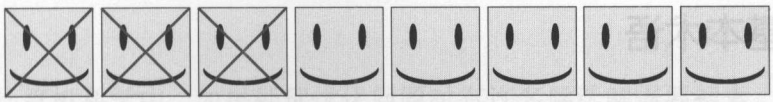
的例子。

在一个遥远的星球上，存在一个物种，它的形状是白方块。

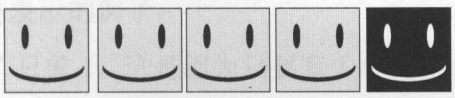


白方块的物种已经和平生活了几千年，直到最近，一个新的物种赶到，即黑圆圈。

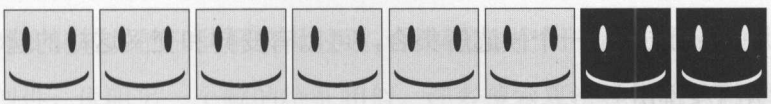
黑圆圈物种是食肉动物，并开始以白方块种群为食。



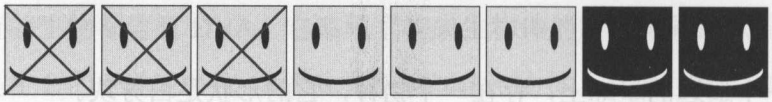
白方块没有任何办法来保护自己并抵抗黑圆圈。直到有一天，一个幸存的白方块随机变异，从一个白方块变成一个黑方块。黑圆圈不再以新的黑方块为食，因为它的颜色和自己一样。



一些幸存的方块种群交配，造就了新一代方块。这些新方块继承了黑方块的颜色基因。



然而，白方块继续被吃掉.....



最后，由于黑方块的进化优势，看起来类似黑圆圈，他们不再被吃了。现在，只剩下黑方块了。

黑方块不再畏惧黑圆圈，它们再一次自由地生活在和平之中。

1.6 基本术语

遗传算法建立在生物进化的概念上，因此，如果你熟悉进化的术语，可能在学习遗传算法时会发现术语有所重叠。这种领域间的相似性是当然的，因为进化算法，确切来说是遗传算法，类似于自然界中发现的过程。

术语

在更深入遗传算法领域之前，我们先了解一些基本的语言和术语，这很重要。随着本书的推进，我们会根据需要引入更复杂的术语。下面是一些较常见的术语的列表，可供参考。

- 种群：这就是一个候选解集合，可以有变异和交叉这样的遗传操作应用于它们。
- 候选解：给定问题的一个可能的解。

- 基因：组成染色体的不可分割的构建块。经典的基因包含 0 或 1。
- 染色体：染色体是一串基因。染色体定义了一个特定的候选解。用二进制编码一个典型的染色体可能包含“01101011”这样的内容。
- 变异：一个过程，其中候选解中的基因被随机改变，以创建新的性状。
- 交叉：其中染色体被组合以创建新的候选解决方案的方法。这有时称为重组。
- 选择：这是选择的候选解，繁殖下一代解的技术。
- 适应度：一个评分，衡量候选解适合给定问题的程度。

1.7 搜索空间

在计算机科学中，如果处理优化问题时有许多候选解需要搜索，我们就称解的集合是“搜索空间”。搜索空间内每个特定的点就是给定问题的一个候选解。在这个搜索空间中有距离的概念，相比位置远离的解，位置彼此靠近的解更可能表现出相似的特征。为了理解这些距离在搜索空间中如何组织，请考虑下面使用二进制遗传表示的例子：

“101”与“111”只差 1。这是因为只要有 1 个变化（0 翻转到 1），就能从“101”变成“111”。这意味着这些解在搜索空间中的空间距离仅为 1。

另一方面，“000”与“111”是有 3 处不同。这就是说距离为 3，在搜索空间“000”与“111”相距为 3。

因为变化较少的一些解彼此较近，所以搜索空间中解的距离可以用来提供一种相似性，说明另一个解的特征相似。许多搜索算法经常将这种理解作为一种策略，以改善搜索结果。

1.7.1 适应度景观

如果搜索空间内发现的候选解标上其个体的适应度水平，我们就可以将搜索空间看成“适应度景观”。图 1-1 提供了一个例子，说明二维适应度景观看起来如何。

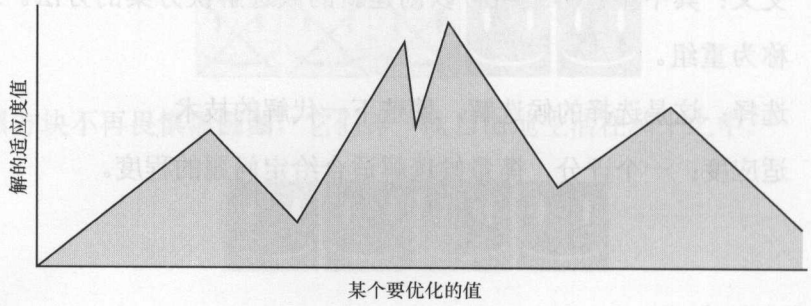


图 1-1 二维适应度景观

适应度景观的横轴是我们要优化的值，竖轴是对应的适应度值。需要指出，这通常是对实际情况的过度简化。大多数真实世界的应用程序都有多个值需要优化，会生成多维适应度景观。

在上面的例子中，可以看到搜索空间中的每个候选解的适应度值。这很容易看到最适应解的位置，但是，要在现实中做到这一点，搜索空间中每个候选解都需要求出适应度函数的值。对于复杂的问题，搜索空间呈指数式增长，计算每个解的适应度值是不合理的。在这种情况下，搜索算法负责找到最佳解的可能位置，同时又受到限制，仅看到搜索空间的一小部分。图 1-2 所示是搜索算法通常会看到什么的一个例子。

请考虑一种算法，它要搜索十亿个（1 000 000 000）可能解构成的搜索空间。即使每个解只需要 1 秒来对适应度求值和赋值，它仍然需要超过 30 年，才能搜索每个可能的解！如果我们不知道搜索空间中每个解的适应度值，我们就无法确切地知道最佳解在哪里。在这种情况下，唯一合理的方法是采用一种

搜索算法，它能在可用的时间内发现足够好的解。在这些条件下，一般来说，遗传算法和进化算法能够非常有效地发现可行的、接近最佳的解。

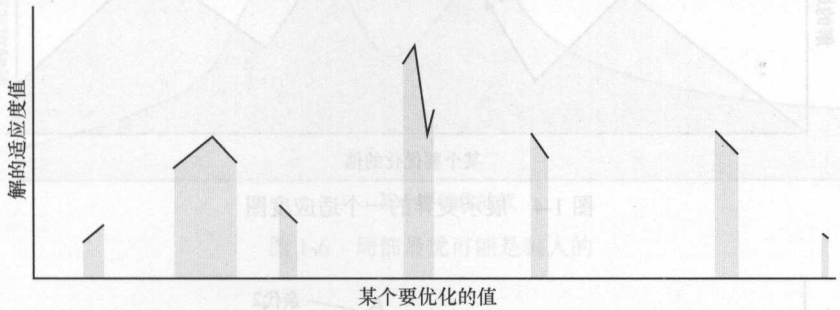


图 1-2　更典型的适应度空间搜索

在搜索空间进行搜索时，遗传算法使用种群的方法。作为其搜索策略的一部分，遗传算法假设两个评分不错的解可以组合，形成一个更适应的后代。这个过程可以在适应度景观（图 1-3）中看出来。

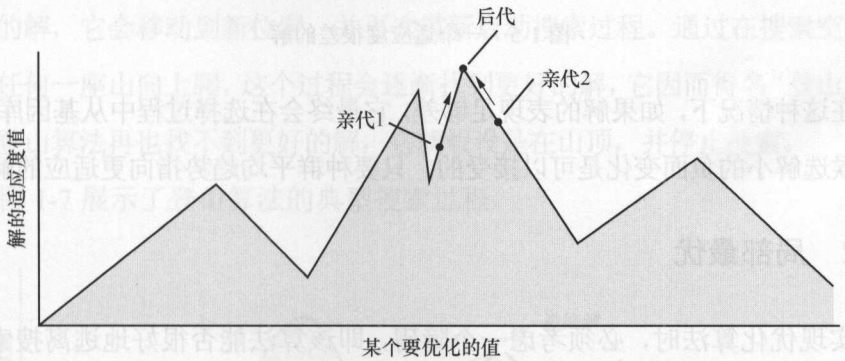


图 1-3　适应度图中的亲代与子代

遗传算法中的变异算子让我们搜索特定候选解的近邻。变异应用于一个基因时，其值随机地改变。这可以表示成在搜索空间（见图 1-4）中跨出一步。

在这两个交叉和变异的例子中，得到的解都有可能比原来亲代的适应度更差（见图 1-5）。

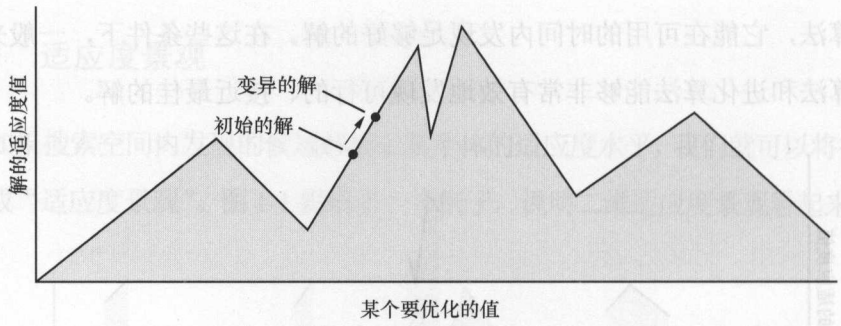


图 1-4 展示变异的一个适应度图

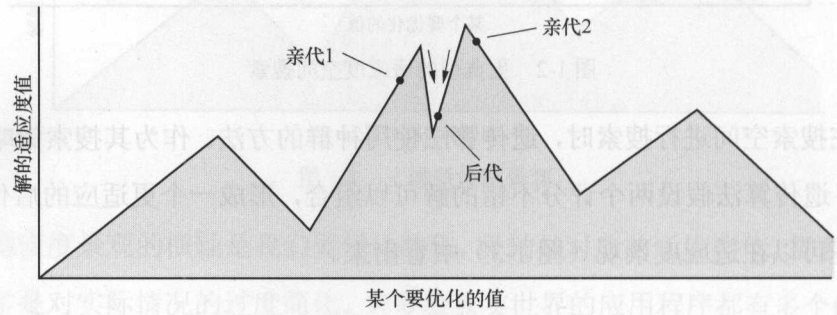


图 1-5 一个适应度很差的解

在这种情况下，如果解的表现足够差，它最终会在选择过程中从基因库删除。个体候选解小的负面变化是可以接受的，只要种群平均趋势指向更适应的解。

1.7.2 局部最优

实现优化算法时，必须考虑一个障碍，即该算法能否很好地逃离搜索空间的局部最优位置。为了更好地表现什么是局部最优，请参考图 1-6。

在这里，我们可以看到适应度景观中的两座小山，它们峰值略微不同。正如前面提到的，优化算法不能够看到整个适应度景观，相反，它能做得最好的是找一些解，它认为这些解很可能处于搜索空间的最佳位置。正是因为这种特点，优化算法通常能在不知不觉中专注于查找搜索空间的次优部分。

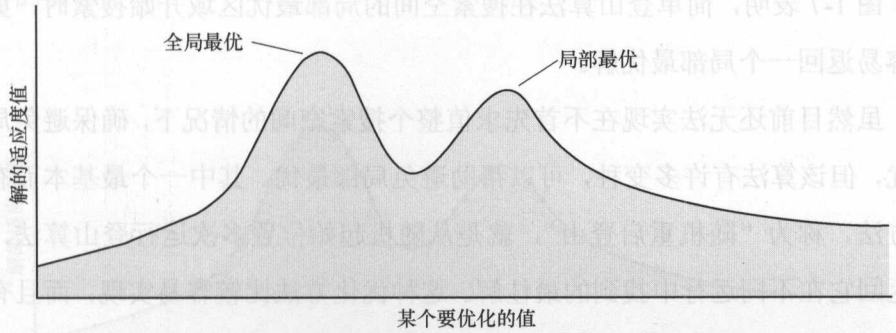


图 1-6 局部最优可能是骗人的

如果实现使用一个简单的登山算法来解决任何足够复杂的问题，这个问题很快就会引起注意。一个简单登山算法没有任何内建的方法来处理局部最优，因此往往会在搜索空间的局部最优区域中终止其搜索。一个简单的随机登山算法相当于没有种群和交叉的遗传算法。该算法相当容易理解，它从搜索空间中的随机点开始，然后评估相邻的解，尝试找到更好的解。如果登山算法在相邻位置找到了更好的解，它会移动到新位置，并再次重新启动搜索过程。通过在搜索空间中找到的任何一座山向上爬，这个过程会逐渐找到更好的解，它因而得名“登山算法”。如果登山算法再也找不到更好的解，它就假设是在山顶，并停止搜索。

图 1-7 展示了登山算法的典型搜索过程。

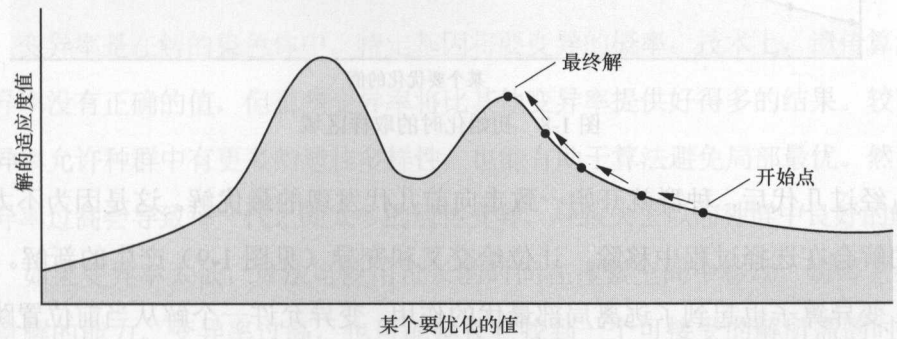


图 1-7 展示登山算法是如何工作的

图 1-7 表明，简单登山算法在搜索空间的局部最优区域开始搜索时，如何很容易返回一个局部最优解。

虽然目前还无法实现在不首先求值整个搜索空间的情况下，确保避免局部最优，但该算法有许多变种，可以帮助避免局部最优。其中一个最基本而有效的方法，称为“随机重启登山”，就是从随机起始位置多次运行登山算法，然后返回它在不同运行中找到的最佳解。这种优化方法比较容易实现，而且有效性令人惊讶。其他方法诸如模拟退火方法［参见 Kirkpatrick，Gelatt，and Vecchi（1983）］和禁忌搜索［参见 Glover（1989）和 Glover（1990）］，它们对登山算法进行了微小改变，都有助于减少局部最优。

在避免局部最优和取得接近最优的解方面，遗传算法的有效性令人惊讶。做到这一点的一种办法，是让种群能够从搜索空间的大片区域取样，定位最佳的区域，继续搜索。图 1-8 展示了种群在初始化时可能如何分布。

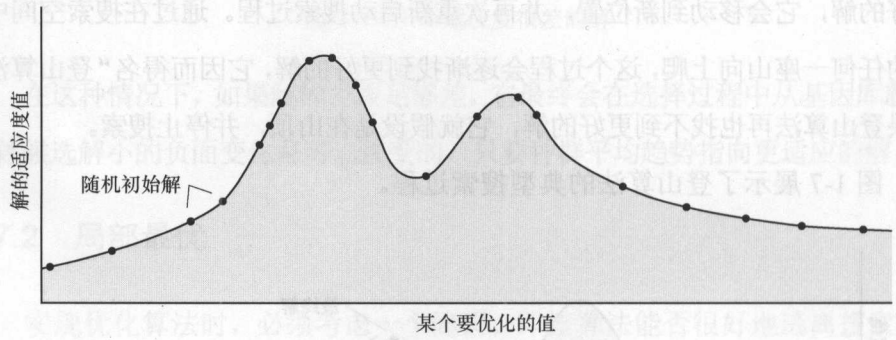


图 1-8 初始化时的取样区域

经过几代后，种群就开始一致走向前几代发现的最优解。这是因为不太适合的解会在选择过程中移除，让位给交叉和变异（见图 1-9）产生的新解。

变异算子也起到了逃离局部最优的作用。变异允许一个解从当前位置跳到搜索空间的另一个位置。这个过程往往会导致在搜索空间的较优区域中发现更

适合的解。

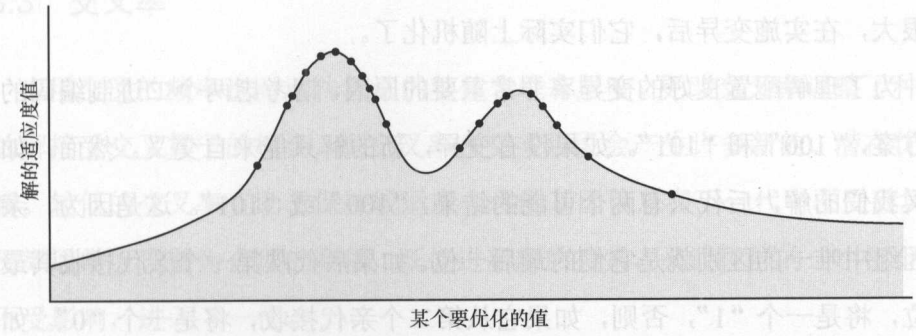


图 1-9 几代后的适应度图已经变化

1.8 参数

虽然所有的遗传算法都基于同样的概念，但它们的具体实现可以差别很大。具体实现不同的一种方式，就是通过它们的参数。一个基本的遗传算法在实现时，至少有几个参数需要考虑。主要的 3 个是变异率、种群规模和交叉率。

1.8.1 变异率

变异率是在解的染色体中，特定基因将要变异的概率。技术上，遗传算法的变异率没有正确的值，但有些变异率将比其他变异率提供好得多的结果。较高的变异率允许种群中有更多的遗传多样性，也能有助于算法避免局部最优。然而，变异率过高会导致每一代之间太多的遗传变异，导致失去以前种群中良好的解。

如果变异率太低，算法可能用过长的时间在搜索空间中移动，妨碍它找到满意解的能力。变异率过高，也可能延长它找到一个可接受的解所需的时间。

虽然高变异率可以帮助遗传算法避免陷入局部最优解，但如果它被设置得太高，就可能对搜索产生负面影响。如前所述，这是由于每一代的解被变异的程度很大，在实施变异后，它们实际上随机化了。

为了理解配置良好的变异率非常重要的原因，请考虑两个二进制编码的候选方案，“100”和“101”。如果没有变异，新的解只能来自交叉。然而，如果交叉我们的解，后代只有两个可能的结果，“100”或“101”。这是因为，亲代基因组中唯一的区别就是它们的最后一位。如果后代从第一个亲代接收其最后一位，将是一个“1”，否则，如果它从第二个亲代接收，将是一个“0”。如果该算法需要找到一个替代解，就需要变异现有的解，得到基因库中没有的新基因信息。

变异率应设置为一个值，该值允许足够的多样性，以防止算法停滞不前，但这个值又不是太高，不会导致该算法失去以前的群体中有价值的遗传信息。这种平衡取决于要解决的问题的性质。

1.8.2 种群规模

种群规模就是遗传算法中任意一代种群的个体数。较大的种群规模，算法可以在搜索空间中取样更多。这将有助于将它导向更准确、全局最优解的方向。小的种群规模通常会导致该算法在搜索空间的局部最优区域发现不太理想的解，但是它们每一代需要的计算资源较少。

这里一样，就像变异率，为了遗传算法的最佳表现，需要找到一种平衡。同样，根据要解决的问题的性质，需要改变种群的规模。为了找到最佳解，山头很多的搜索空间通常需要较大的种群规模。有趣的是，选择种群规模时有一个点，超过这个点，增加规模也不会为算法提供多少改进以找到精度更好的解。相反，由于需要额外的计算来处理增加的个体，它会让执行变慢。在这个转折

点附近的种群规模，通常提供了资源和结果之间的最佳平衡。

1.8.3 交叉率

应用交叉的频率也对遗传算法的整体表现有影响。更改交叉率调整了种群中的解接受交叉算子的机会。高交叉率在交叉阶段会产生许多新的、潜在优越的解。较低的交叉率有助于保持较适应个体的基因信息在下一代中不受破坏。交叉率通常应设置为合理的高低，既促进新解的搜索，又让种群的一小部分保持不受影响，进入下一代。

1.9 基因表示

除了这些参数，影响遗传算法表现的另一个因素是所用的基因表示。这是遗传信息在染色体内的编码方式。更好的表示会让解的编码方式既表现力强，又容易进化。Holland（1975）的遗传算法是基于二进制基因表示。他建议染色体用含有 0 和 1 的序列。这种二进制表示可能是目前最简单的编码，但对于很多问题表现力不是很够，不是合适的首选。请考虑一个例子，用二进制表示来编码一个整数，它将被优化，用于某函数。在这个例子中，“000”代表 0，而“111”代表 7，就像通常二进制一样。如果染色体的第一个基因变异（一位从 0 翻转到 1，或从 1 到 0），它的编码值将改变 4（“111”=7，“011”=3）。但如果染色体的最后一个基因变异，编码值将改变 1（“111”=7，“110”=6）。在这里，根据正在作用于染色体的是哪个基因，变异算子对候选解有不同的效果。这种差异是不理想的，因为它会降低算法的表现和可预见性。对于这个例子，如果采用带有互补变异算子的整数，该算子对基因值的改变较小，就会更好。

除了可以使用简单的二进制表示和整数，遗传算法还可以使用浮点数、基于树的表示、对象和遗传编码所需的任何其他数据结构。如果要建立一个有效的遗传算法，选择正确的表示是关键。

1.10 终止

遗传算法可以继续进化出新的候选解，无论需要多长时间。根据问题的性质，遗传算法的运行时间可以从几秒钟到几年！我们将遗传算法完成搜索的条件称为终止条件。

一些典型的终止条件是：

- 到达世代的最大数目；
- 超过分配给它的时间；
- 发现一个满足所需条件的解；
- 该算法已经达到了一个稳定阶段。

有时也许最好有多个终止条件。例如，如果设置最大的时间限制，并在找到适当的解时能提前终止，就非常方便。

1.11 搜索过程

作为本章的结束，让我们一步一步地了解遗传算法背后的基本过程，如图 1-10 所示。

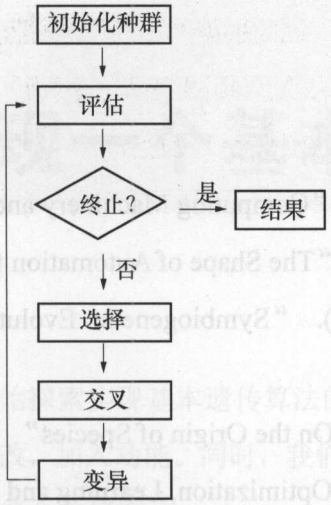


图 1-10 一般遗传算法的过程

1. 遗传算法开始，初始化候选解的种群。这通常是随机提供整个搜索空间的均匀覆盖。

2. 接下来，通过为种群中的每个个体分配一个适应度值，对种群进行评估。在这个阶段，我们常常要注意当前最优解，以及种群的平均适应度。

3. 评估后，根据终止条件集，该算法决定它是否应该终止搜索。通常这是因为该算法已达到指定的世代数量，或已经找到适当的解。

4. 如果终止条件不满足，种群经过一个选择阶段，基于适应度评分，从种群中选择个体。适应度越高，个体就更有机会被选择。

5. 下一阶段对选择的个体应用交叉和变异。这个阶段为下一代创建新个体。

6. 此时新种群返回到评估步骤，过程重新开始。我们称这种循环的每一圈为一个世代。

7. 如果终止条件最终满足，算法会跳出循环，通常向用户返回最后的搜索结果。

1.12 参考文献

Turing, A.M. (1950). “Computing Machinery and Intelligence”

Simon, H.A. (1965). “The Shape of Automation for Men and Management”

Barricell, N.A. (1975). “Symbiogenetic Evolution Processes Realised by Artificial Methods”

Darwin, C. (1859). “On the Origin of Species”

Dorigo, M. (1992). “Optimization, Learning and Natural Algorithms”

Rechenberg, I. (1965) “Cybernetic Solution Path of an Experimental Problem”

Rechenberg, I. (1973) “Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution”

Schwefel, H.-P. (1975) “Evolutionsstrategie und numerische Optimierung”

Schwefel, H.-P. (1977) “Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie”

Fogel L.J; Owens, A.J; and Walsh, M.J. (1966) “Artificial Intelligence through Simulated Evolution”

Holland, J.H. (1975) “Adaptation in Natural and Artificial Systems”

Dorigo, M. (1992) “Optimization, Learning and Natural Algorithms”

Glover, F. (1989) “Tabu search. Part I”

Glover, F. (1990) “Tabu search. Part II”

Kirkpatrick, S; Gelatt, C.D, Jr., and Vecchi, M.P. (1983) “Optimization by simulated annealing”

第2章

实现一个基本遗传算法

在本章中，我们将开始探索实现基本遗传算法的技术。本章开发的程序，将在后面的章节中进行修改，加入功能。同时，我们也将探讨基于遗传算法的参数和配置，以及它的性能会如何变化。

要尝试运行本节中的代码，需要先在计算机上安装 Java JDK。你可以从 Oracle 的网站上免费下载并安装 Java JDK：

```
oracle.com/technetwork/java/javase/downloads/index.html
```

除了安装 Java JDK，你也可以选择安装支持 Java 的 IDE，比如 Eclipse 或 NetBeans，虽然这不是必需的。

2.1 实现之前

实现遗传算法之前，最好是先考虑遗传算法是不是完成手上任务的正确途径。往往有更好的技术来解决具体的优化问题，通常是利用一些基于领域的启发式搜索。遗传算法是与领域无关的，或称为“弱方法”，能够应用该方法的

问题，不需要任何特定的先验知识来协助其搜索过程。由于这个原因，如果没有任何已知的领域特定的知识可用于帮助引导搜索过程，遗传算法仍然可以用来发现潜在的解。

如果已经确定弱搜索方法是合适的，还应该考虑采用的弱方法的类型。这可能只是因为替代方法提供了更好的平均结果，也可能是因为替代方法更容易实现，需要的计算资源较少，或可以在较短的时间内找到足够好的结果。

2.2　基本遗传算法的伪代码

基本遗传算法的伪代码如下：

```
1: generation = 0;
2: population[generation] = initializePopulation(populationSize);
3: evaluatePopulation(population[generation]);
3: While isTerminationConditionMet() == false do
4:     parents = selectParents(population[generation]);
5:     population[generation+1] = crossover(parents);
6:     population[generation+1] = mutate(population[generation+1]);
7:     evaluatePopulation(population[generation]);
8:     generation++;
9: End loop;
```

伪代码从创建遗传算法的初始种群开始。然后，对这个群体进行评估，求出其个体的适应度值。下一步，检查确定遗传算法的终止条件是否已经满足。

如果未满足，遗传算法开始循环，种群经过第一轮的交叉和变异，然后重新评估。此后，持续进行交叉和变异，直到满足终止条件，遗传算法终止。

这段伪代码展示了遗传算法的基本过程，但我们有必要仔细查看每个步骤，充分了解如何创建一个令人满意的遗传算法。

2.3 关于本书的代码示例

本书中的每一章都作为一个包，放在附带的 Eclipse 项目中。每个包都至少有 4 个类。

- GeneticAlgorithm 类，它抽象了遗传算法本身，为接口方法提供了针对问题的实现，如交叉、变异、适应度评估和终止条件检查。
- Individual 类，它表示单个候选解及其染色体。
- Population 类，它表示一个种群或个体的一个世代，并对它们应用群组级别的操作。
- 包含 main 方法的类，包括一些引导代码，前面伪代码的具体版本，以及具体问题可能需要的任何辅助工作。这些类根据它解决的问题来命名，如 AllOnesGA、RobotController 等。

在本章开始时写下的 GeneticAlgorithm、Population 和 Individual 类，需要针对本书后面的各章进行修改。

你可以认为这些类实际上都是接口的具体实现，如 GeneticAlgorithmInterface、PopulationInterface 和 IndividualInterface，但是为了让 Eclipse 项目的布局保持简单，我们没有使用接口。

本书中的 GeneticAlgorithm 类总是实现了一些重要的方法，如 calcFitness、evalPopulation、isTerminationConditionMet、crossoverPopulation 和 mutatePopulation。但是，根据手上问题的要求，这些方法的内容在每章中略有不同。

在尝试本书中的例子时，我们建议针对每个新问题复制 GeneticAlgorithm、Population 和 Individual 类，因为一些方法的实现在各章中保持不变，但另一些方法会有所不同。

此外，请务必阅读附带 Eclipse 项目源代码中的注释！为了在本书中节省篇幅，我们已经省略了较长的注释和文档注释块，但非常认真地在可供下载的 Eclipse 文件中提供了充分的源代码注释。对你来说，这就像读第二本书一样！

在许多情况下，本书的章节会要求你在一个类中添加或修改一个方法。一般情况下，在文件的什么位置添加新方法并不重要，所以在这些情况下，我们要么在例子中省略类的其余部分，要么只显示函数签名，仅仅是帮助你保持正确的方向。

2.4 基本实现

为了消除所有不必要的细节，保持最初的实现容易尝试，本书中介绍的第一个遗传算法将是简单的二进制遗传算法。

二进制遗传算法比较容易实现，对于解决许多种优化问题，它可能是非常有效的工具。你可能还记得第 1 章提到，二进制遗传算法是由 Holland（1975）提出的原创的遗传算法。

2.4.1 问题

首先，让我们回顾一下“全一”问题，它是可以用二进制遗传算法来解决的一个非常基本的问题。

该问题不是很有趣，但作为一个简单的问题，它的作用是强调所涉及的基本技术。顾名思义，该问题就是发现全部由 1 构成的字符串。因此，对于长度为 5 的字符串，最优解是“11111”。

2.4.2 参数

既然有了要解决的问题，让我们继续研究实现。我们要做的第一件事就是建立遗传算法参数。如前所述，这 3 个主要参数是种群规模、变异率和交叉率。本章中，我们还引入了一个名为“精英主义（elitism）”的概念，并将它作为遗传算法的参数之一。

首先，创建一个名为 GeneticAlgorithm 的类。如果你使用 Eclipse，可以通过选择 File ➤ New ➤ Class 来做到这一点。在本书中，我们选择用对应的章名来命名包，所以我们会在包“Chapter2”中工作。

这个 GeneticAlgorithm 类将包含遗传算法本身的操作所需的方法和变量。例如，这个类包括处理交叉、变异、适应度评估和终止条件检查的逻辑。该类创建后，添加一个构造方法，它接受 4 个参数：种群规模、变异率、交叉率和精英成员数。

```
package chapter2;
/**
 * Lots of comments in the source that are omitted here!
 */
public class GeneticAlgorithm {
```

```
    private int populationSize;
    private double mutationRate;
    private double crossoverRate;
    private int elitismCount;

public GeneticAlgorithm(int populationSize, double mutationRate, double
crossoverRate, int elitismCount) {
        this.populationSize = populationSize;
        this.mutationRate = mutationRate;
        this.crossoverRate = crossoverRate;
        this.elitismCount = elitismCount;
    }

    /**
     * Many more methods implemented later...
     */
}
```

传入所需的参数，这个构造方法将利用所需的配置，创建 GeneticAlgorithm 类的新实例。

现在，我们应该创建引导类：回想一下，每章都需要一个引导类，用于初始化遗传算法，并作为应用程序的起点。将该类命名为“AllOnesGA”，并定义一个 main 方法：

```
package chapter2;
public class AllOnesGA {
    public static void main(String[] args) {
        // Create GA object
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.95, 0);
```

```
            // We'll add a lot more here...
        }
    }
```

暂时，我们就用一些典型的参数值：种群规模=100，变异率=0.01，交叉率=0.95，精英计数为 0（实际上暂且禁用它）。在本章结束，当你已完成了以上的内容时，可以尝试更改这些参数，看看它们如何影响算法的表现。

2.4.3 初始化

下一步就是初始化一个潜在解构成的种群。这通常是随机的，但偶尔也可能采用系统化的方法会更好，可以利用对搜索空间的已知信息来初始化种群。在这个例子中，种群中每个个体将随机初始化。我们可以为染色体的每个基因随机选择 1 或 0，实现这一点。

初始化种群之前，我们需要创建两个类，一个管理并创建种群，另一个管理和创建种群的个体。这此类包含一些方法，例如获取个体适应度，或在种群中取得最适应的个体。

首先，让我们从创建 Individual 类开始。请注意，为了节省篇幅，我们省略了所有注释和方法的文档注释块！你可以在附带的 Eclipse 项目中找到该类的完全注释版本。

```
package chapter2;

public class Individual {
    private int[] chromosome;
    private double fitness = -1;
```

```
public Individual(int[] chromosome) {
        // Create individual chromosome
        this.chromosome = chromosome;
}

public Individual(int chromosomeLength) {
        this.chromosome = new int[chromosomeLength];
        for (int gene = 0; gene < chromosomeLength; gene++) {
                if (0.5 < Math.random()) {
                        this.setGene(gene, 1);
                } else {
                        this.setGene(gene, 0);
                }
        }
}
public int[] getChromosome() {
        return this.chromosome;
}
public int getChromosomeLength() {
        return this.chromosome.length;
}
public void setGene(int offset, int gene) {
        this.chromosome[offset] = gene;
}
public int getGene(int offset) {
        return this.chromosome[offset];
}
public void setFitness(double fitness) {
```

```
        this.fitness = fitness;
    }
    public double getFitness() {
        return this.fitness;
    }
    public String toString() {
        String output = "";
        for (int gene = 0; gene < this.chromosome.length; gene++) {
            output += this.chromosome[gene];
        }
        return output;
    }
}
```

Individual 类代表一个候选解，主要负责存储和操作一条染色体。请注意，Individual 类也有两个构造方法。一个构造方法接受一个整数（代表染色体的长度），在初始化对象时将创建一条随机的染色体。另一个构造方法接受一个整数数组，用它作为染色体。

除了管理 Individual 的染色体，它也追踪个体的适应度值，也知道如何将自己打印为一个字符串。

下一步骤是创建 Population 类，它提供管理群体中一组个体所需的功能。

像往常一样，注释和文档注释块在这一章中已经省略了，一定要看看 Eclipse 项目，了解更多背景！

```
package chapter2;

import java.util.Arrays;
import java.util.Comparator;
```

```
public class Population {
    private Individual population[];
    private double populationFitness = -1;

    public Population(int populationSize) {
        this.population = new Individual[populationSize];
    }

    public Population(int populationSize, int chromosomeLength) {
        this.population = new Individual[populationSize];

        for (int individualCount = 0; individualCount <
        populationSize; individualCount++) {
            Individual individual = new
            Individual(chromosomeLength);
            this.population[individualCount] = individual;
        }
    }

    public Individual[] getIndividuals() {
        return this.population;
    }

    public Individual getFittest(int offset) {
        Arrays.sort(this.population, new Comparator<Individual>() {
            @Override
            public int compare(Individual o1, Individual o2) {
```

```
                if (o1.getFitness() > o2.getFitness()) {
                    return -1;
                } else if (o1.getFitness() < o2.getFitness()) {
                    return 1;
                }
                return 0;
            }
        });
        return this.population[offset];
    }

    public void setPopulationFitness(double fitness) {
        this.populationFitness = fitness;
    }

    public double getPopulationFitness() {
        return this.populationFitness;
    }

    public int size() {
        return this.population.length;
    }

    public Individual setIndividual(int offset, Individual individual) {
        return population[offset] = individual;
    }

        public Individual getIndividual(int offset) {
        return population[offset];
```

```
        }

        public void shuffle() {
            Random rnd = new Random();
            for (int i = population.length - 1; i > 0; i--) {
                int index = rnd.nextInt(i + 1);
                Individual a = population[index];
                population[index] = population[i];
                population[i] = a;
            }
        }
    }
```

Population 类相当简单，它的主要功能是保存由个体构成的一个数组，能够通过类方法方便地访问。例如 getFittest()和 setIndividual()这样的方法，能够访问并更新种群中的个体。除了保存个体，它也存储了种群的总体适应度，在稍后实现选择方法时，这很重要。

既然我们有了种群和个体类，就可以在 GeneticAlgorithm 类中将它们实例化。要做到这一点，只需创建一个名为“initPopulation”方法，放在 GeneticAlgorithm 类中的任意位置。

```
public class GeneticAlgorithm {

    /**
     * The constructor we created earlier is up here...
     */

    public Population initPopulation(int chromosomeLength) {
        Population population = new Population(this.populationSize,
```

```
                chromosomeLength);
                return population;
        }
        /**
         * We still have lots of methods to implement down here...
         */
}
```

既然有了 Population 和 Individual 类，我们就可以回到 AllOnesGA 类，开始使用 initPopulation 方法。回想一下，AllOnesGA 类只有一个 main 方法，而且它代表本章前面介绍的伪代码。

在 main 方法中初始化种群时，还需要指定个体染色体的长度，这里，我们取长度为 50：

```
public class AllOnesGA {
   public static void main(String[] args){
        // Create GA object
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.95, 0);

        // Initialize population
        Population population = ga.initPopulation(50);
   }
}
```

2.4.4 评估

在评估阶段，种群中的每个个体都计算其适应度值，并存储以便将来使用。为了计算个体的适应度，我们使用所谓的“适应度函数”。

遗传算法通过选择来引导进化过程，得到更好的个体。因为正是适应度函数使得这种选择成为可能，所以适应度函数应该设计良好，提供个体适应度的准确值，这很重要。如果适应度函数设计得不够好，可能需要更长的时间才能找到满足的最低标准的解，也可能根本找不到可以接受的解。

适应度函数往往是遗传算法中需要最多计算的组件。正因为如此，适应度函数做好优化，有助于预防瓶颈，让算法高效地运行。这很重要。

每个特定的优化问题，都需要一个特别开发的适应度函数。在“全一”问题的例子中，适应度函数相当简单，只需计算个体染色体中 1 的数量。

现在为 GeneticAlgorithm 类添加一个 calcFitness 方法。该方法将计算染色体中 1 的个数，然后除以染色体的长度，使输出规格化，在 0 和 1 之间。你可以在 GeneticAlgorithm 类的任意位置添加此方法，因此下面省略了其周围的代码：

```
public double calcFitness(Individual individual) {

    // Track number of correct genes
    int correctGenes = 0;

    // Loop over individual's genes
    for (int geneIndex = 0; geneIndex < individual.getChromosomeLength();
    geneIndex++) {
        // Add one fitness point for each "1" found
        if (individual.getGene(geneIndex) == 1) {
            correctGenes += 1;
        }
    }

    // Calculate fitness
```

```
        double fitness = (double) correctGenes / individual.
        getChromosomeLength();

        // Store fitness
        individual.setFitness(fitness);

        return fitness;
    }
```

我们也需要一个简单的辅助方法，遍历每个个体并评估它们（即对每个个体调用 calcFitness）。我们称这个方法为 evalPopulation，并将它也添加到 GeneticAlgorithm 类中。它看起来应该像下面这样，同样可以将它添加在任何位置：

```
public void evalPopulation(Population population) {
        double populationFitness = 0;

        for (Individual individual : population.getIndividuals()) {
            populationFitness += calcFitness(individual);
        }

        population.setPopulationFitness(populationFitness);
}
```

此时，GeneticAlgorithm 类应该具有以下方法。简洁起见，我们省略了函数体，只是显示类的折叠视图：

```
package chapter2;

public class GeneticAlgorithm {
```

```
    private int populationSize;
    private double mutationRate;
    private double crossoverRate;
    private int elitismCount;

    public GeneticAlgorithm(int populationSize, double mutationRate,
    double crossoverRate, int elitismCount) { }
    public Population initPopulation(int chromosomeLength) { }
    public double calcFitness(Individual individual) { }
    public void evalPopulation(Population population) { }
}
```

如果缺少其中任何属性或方法，请现在就回去实现它们。我们还有 4 个方法要在 GeneticAlgorithm 类中实现：isTerminationConditionMet、selectParent、crossoverPopulation 和 mutatePopulation。

2.4.5　终止检查

接下来需要检查终止条件是否已经满足。有许多不同类型的终止条件。有时可能知道最佳解是什么（更确切地说，是可能知道最佳解的适应度值），在这种情况下，我们可以直接检查正确解。然而，并非总是能够知道最佳解的适应度是什么，所以我们可以在解变得“足够好”时终止，即解超过某个适应度阈值。如果算法运行了太长时间（太多世代），我们也可以终止，或者可以结合一些因素，决定终止该算法。

由于“全一”问题很简单，事实上，我们知道正确的适应度应该是 1，在这种情况下，找到正确的解就终止，这是合理的。但情况并非总是这样！事实上，很少会这样。但我们很幸运，这是一个简单的问题。

首先，我们必须构建一个函数，检查终止条件是否已发生。我们可以在GeneticAlgorithm 类中添加如下代码来实现这一点。代码添加在任意位置，和往常一样，为了简洁，我们省略了其他的类。

```
public boolean isTerminationConditionMet(Population population) {
    for (Individual individual : population.getIndividuals()) {
        if (individual.getFitness() == 1) {
            return true;
        }
    }

    return false;
}
```

上述方法检查种群中的每个个体，如果种群中任何个体的适应度为 1，就返回 true（这表明，我们已经找到了一个终止条件，可以停止）。

既然已经建立了终止条件，就可以在 AllOnesGA 类的主引导方法中添加一个循环，并使用新添加的终止检查作为其循环条件。如果终止检查返回 true，遗传算法将停止循环，并返回其结果。

为了创建进化循环，将执行类 AllOnesGA 的 main 方法修改为如下所示。下面代码片段的前两行已经在 main 方法中。通过添加这段代码，我们将继续实现本章开头展示的伪代码：回忆一下，main 方法是遗传算法伪代码的具体表现。下面是 main 方法现在的样子：

```
public static void main(String[] args) {
    // These two lines were already here:
    GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001, 0.95, 0);
     Population population = ga.initPopulation(50);

     // The following is the new code you should be adding:
```

```
        ga.evalPopulation(population);
        int generation = 1;

        while (ga.isTerminationConditionMet(population) == false) {
            // Print fittest individual from population
            System.out.println("Best solution: " + population.
            getFittest(0).toString());

            // Apply crossover
            // TODO!

            // Apply mutation
            // TODO!

            // Evaluate population
            ga.evalPopulation(population);

            // Increment the current generation
            generation++;
        }

        System.out.println("Found solution in " + generation + "
        generations");
        System.out.println("Best solution: " + population.getFittest(0).
        toString());
    }
```

我们添加了一个进化循环，检查 isTerminationConditionMet 的输出。main 方法的新内容还包括在循环之前和循环之内添加了 evalPopulation 调用，用于追踪世代数目的 generation 变量，以及调试消息，以便让你知道每一代的最佳解是怎样的。

我们还增加了程序结束时的代码：循环退出时，打印关于最终解的一些信息。

然而，现在我们的遗传算法会运行，但不会永远进化！我们将陷入一个无限循环中，除非我们很幸运，随机产生的一个个体恰好是“全一”。你可以点击 Eclipse 中的“Run”按钮，直接看到这样的行为。相同的解将一遍又一遍地提交，循环永远不会结束。你不得不点击 Eclipse 控制台上方的“Terminate”按钮，强制程序停止。

为了继续建立我们的遗传算法，需要实现另外两个概念：交叉和变异。通过随机变异和最适者生存，这些概念实际上推动了种群向前进化。

2.4.6 交叉

现在，是时候开始运用变异和交叉来进化种群了。交叉算子是一个过程，在这个过程中，种群中的个体交换它们的遗传信息，希望创建一个新的个体，包含亲代基因组中最好的部分。

在交叉过程中，考虑种群中每个个体是否参与交叉，这时使用交叉率参数。通过比较交叉率和一个随机数，我们可以决定个体是应该参与交叉，还是应该直加入下一个种群，不受交叉影响。如果选择了一个个体参与交叉，就需要找到第二个亲代。要找到第二个亲代，我们需要在多种可能的选择方法中挑一种。

2.5 轮盘赌选择

轮盘赌选择（也称为适应度比例选择）是用轮盘赌为类比，从种群中选择个体的方法。这种想法是根据种群中个体的适应值，将它们放置在一个假想的

轮盘上。个体的适应度越高，在轮盘上占据的空间就越多。图 2-1 展示了在这个过程中，个体通常如何放置。

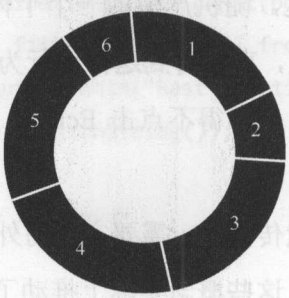


图 2-1　轮盘赌

上面轮盘上的每个数字表示种群中的一个个体。该个体的适应度越高，它们占轮盘赌的部分就越大。如果你现在想象轮盘转动，适合度更高的个体就更可能被选中，因为它们占据了轮盘的更多空间。这就是为什么这种选择方法通常称为适应度比例选择，因为解的选择是基于它们的适应度与种群中其他个体适应度的比例的。

我们还可以用许多其他选择方法，例如锦标赛选择（第 3 章）和随机通用采样（适应度比例选择的一种高级形式）。然而，在本章中，我们将实现最常见的选择方法之一：轮盘赌选择。在后面的章节中，我们将查看其他选择方法，以及它们的差异。

2.6　交叉方法

在交叉过程中，除了用不同的选择方法，还有可用不同的方法在两个个体之间交换遗传信息。不同的问题具有不太一样的特点，采用特定的交叉方法更

好。例如，“全一”问题只要求完全由 1 构成的字符串。字符串“00111”与字符串“10101”具有相同的适应度值，因为它们都包含 3 个 1。对于这种类型的遗传算法，情况并非总是如此。设想我们试图创建一个字符串，它按顺序列出数字 1 到 5。在这种情况下，字符串“12345”与“52431”的适应度值非常不同。这是因为我们不只是寻求正确的数字，也寻求正确的顺序。对于这样的问题，适合采用考虑基因顺序的交叉方法。

我们在这里实现的交叉方法是均匀交叉（uniform crossover）。在这种方法中，后代的每个基因都有 50%的机会来自第一个亲代或其第二个亲代，如图 2-2 所示。

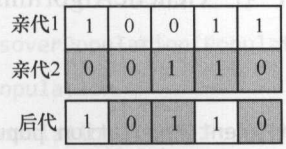


图 2-2 均匀交叉方法

2.7 交叉伪代码

既然有了选择和交叉的方法，让我们来看一些伪代码，其中概述了要实现的交叉过程。

```
1: For each individual in population:
2:     newPopulation = new array;
2:       If crossoverRate > random():
3:           secondParent = selectParent();
4:         offspring = crossover(individual, secondParent);
5:         newPopulation.push(offspring);
```

```
6:      Else:
7:          newPopulation.push(individual);
8:      End if
9: End Loop;
```

2.8 交叉实现

为了实现轮盘赌选择，在 GeneticAlgorithm 类的任意位置增加一个 selectParent( )方法。

```
public Individual selectParent(Population population) {
        // Get individuals
        Individual individuals[] = population.getIndividuals();

        // Spin roulette wheel
        double populationFitness = population.getPopulationFitness();
        double rouletteWheelPosition = Math.random() * populationFitness;

        // Find parent
        double spinWheel = 0;
        for (Individual individual : individuals) {
                spinWheel += individual.getFitness();
                if (spinWheel >= rouletteWheelPosition) {
                        return individual;
                }
        }
```

```
        return individuals[population.size() - 1];
    }
```

selectParent( )方法基本上是反着玩轮盘赌。在赌场，轮盘上已经有标记，然后旋转的轮盘，等待球落入位置。但在这里，我们先选择一个随机位置，然后反向工作，弄清楚哪个个体位于该位置。在算法上，这样更容易实现。选择一个介于 0 和种群总适应度的随机数，然后逐个检查每个个体，同时累加它们的适应度值，直到达到起初选择的随机位置。

既然已经添加了选择方法，下一步就是创建一个交叉方法，利用这个 selectParent() 方法来选择交叉伙伴。首先，将下面的交叉方法添加到 GeneticAlgorithm 类。

```
public Population crossoverPopulation(Population population) {
        // Create new population
        Population newPopulation = new Population(population.size());

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size();
        populationIndex++) {
                Individual parent1 = population.getFittest(populationIndex);

                // Apply crossover to this individual?
                if (this.crossoverRate > Math.random() && populationIndex >
                this.elitismCount) {
                        // Initialize offspring
                        Individual offspring = new Individual(parent1.
                        getChromosomeLength());

                        // Find second parent
                        Individual parent2 = selectParent(population);
```

```
                // Loop over genome
                for (int geneIndex = 0; geneIndex < parent1.
                getChromosomeLength(); geneIndex++) {
                    // Use half of parent1's genes and half of
                       parent2's genes
                    if (0.5 > Math.random()) {
                        offspring.setGene(geneIndex,
                        parent1.getGene(geneIndex));
                    } else {
                        offspring.setGene(geneIndex,
                        parent2.getGene(geneIndex));
                    }
                }

                // Add offspring to new population
                newPopulation.setIndividual(populationIndex,
                offspring);
            } else {
                // Add individual to new population without applying
                crossover
                newPopulation.setIndividual
                (populationIndex, parent1);
            }
        }

        return newPopulation;
    }
```

在 crossoverPopulation()方法的第一行，为下一代创建一个新的空种群。接下来，遍历种群，利用交叉率来考虑每个个体的交叉（这里还有一个神秘的术语“精英主义”，我们将在下一节讨论）。如果个体不经过交叉，它就直接加入

下一个种群，否则就创建一个新个体。后代染色体的填充方法是遍历亲代染色体，随机从每个亲代选择基因，加入后代的染色体。针对种群的每个个体完成这个交叉过程后，交叉方法返回下一代的种群。

至此，我们可在以 AllOnesGA 类的 main 方法中，实现交叉的功能。整个 AllOnesGA 类和 main 方法打印如下，但和以前比，唯一的变化是在“Apply crossover”注释下面，加入一行 crossoverPopulation()调用。

```
package chapter2;
public class AllOnesGA {
    public static void main(String[] args) {
        // Create GA object
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001, 0.95, 0);

        // Initialize population
        Population population = ga.initPopulation(50);

        // Evaluate population
        ga.evalPopulation(population);

        // Keep track of current generation
        int generation = 1;

        while (ga.isTerminationConditionMet(population) == false) {
            // Print fittest individual from population
            System.out.println("Best solution: " + population.
            getFittest(0).toString());

            // Apply crossover
            population = ga.crossoverPopulation(population);
```

```
            // Apply mutation
            // TODO

            // Evaluate population
            ga.evalPopulation(population);

            // Increment the current generation
            generation++;
        }

        System.out.println("Found solution in " + generation + "
        generations");
        System.out.println("Best solution: " + population.
        getFittest(0).toString());
    }
}
```

现在，运行程序应该能工作，并返回一个有效的解！你可以尝试一下，点击 Eclipse 中的 Run 按钮，观察控制台中的结果。

正如你所看到的，单独的交叉足以进化种群。然而，如果没有变异，遗传算法很容易陷入局部最优而不能找到全局最优。在这样简单的问题中，我们不会看到这种现象，但在更复杂的问题领域，我们需要某种机制，将一个种群推离局部最优，试试看是否有其他更好的解。这就是变异的随机性发挥作用的地方：如果一个解停滞在局部最优，随机事件可能将它踢向正确的方向，让它朝着更好的解前进。

2.8.1　精英主义

在讨论变异之前，先来看看我们在交叉方法中引入的 elitismCount 参数。

由于交叉和变异算子的效果，基本遗传算法往往会在世代之间失去种群中最好的个体。然而，我们需要这些算子来找到更好的解。要在实战中看到这个问题，只需编辑遗传算法的代码，针对每一代打印出最适应个体的适应度。你会发现，尽管它通常会增长，有时候在交叉和变异时最优解会丢失，代之以非最优解。

解决这个问题的一个简单的优化技术，就是始终让最适合的个体不作改变，添加到下一代的种群中。这样，最优个体不会在世代之间丢失。虽然这些个体没有进行交叉操作，但它们仍会被选为另一个个体的亲代，让它们的遗传信息仍然可以和种群中的其他个体分享。将最优解保留到下一代的过程称为“精英主义”。

通常情况下，群体中“精英”个体的最佳数目只占种群总规模的很小一部分。原因是如果该值太高，就会减缓遗传算法搜索过程，因为保留太多个体导致缺乏遗传多样性。和以前讨论的其他参数类似，为最佳表现找到一种平衡，这是非常重要的。

实现精英主义在交叉和变异中都很简单。让我们重新检查 crossover Population( )，看看是否应该应用交叉：

```
// Apply crossover to this individual?
if (this.crossoverRate > Math.random() && populationIndex >= this.
elitismCount) {
      // ...
}
```

仅当交叉条件满足且个体不是精英，才交叉。

个体如何才算精英？此时，种群中的个体已经按它们的适应度排序，因此最强大的个体索引值最小。因此，如果我们需要 3 个精英个体，应该跳过索引

0～2。这将保留最强大的个体，让它们保持不变，传递到下一代。在接下来的变异代码中，我们将使用完全相同的条件。

2.8.2 变异

要完成进化过程，最后要添加的是变异。像交叉一样，有许多不同的变异方法可供选择。如果使用二进制串，一种比较常见的方法是所谓的“位翻转”变异。你可能已经猜到，位翻转变异就是根据位的初始值，将它从 1 翻转为 0，或从 0 翻转到 1。如果染色体使用另外某种形式编码，通常会采用不同的变异方法，以便更好地利用这种编码。

在选择变异和交叉方法时，有一点非常重要，即确保你选择的方法仍然得到一个有效解。我们将后续章节中看到这一概念的实战，但对于这个问题，我们只需要确保基因变异只能产生 0 和 1。例如，基因变异为 7 将给出一个无效的解。

在本章中，这个建议似乎没有实际意义，而且过于明显，但请考虑另一个简单的问题，如果你需要不重复地从 1 到 6 排序（即得到“123456”）。变异算法如果简单地选择 1 到 6 之间的随机数，可能产生“126456”，使用“6”两次，这将是一个无效的解，因为每个数字只能用一次。正如你所看到的，即使简单的问题，有时也需要复杂的技术。

与交叉类似，变异基于变异率应用于个体。如果变异率设定为 0.1，那么每个基因在变异阶段发生变异的机会是 10%。

让我们继续，为 GeneticAlgorithm 类添加变异的功能。可以将它添加到任何位置：

```
public Population mutatePopulation(Population population) {
        // Initialize new population
```

```
Population newPopulation = new Population(this.populationSize);

// Loop over current population by fitness
for (int populationIndex = 0; populationIndex < population.size();
populationIndex++) {
        Individual individual = population.
        getFittest(populationIndex);

        // Loop over individual's genes
        for (int geneIndex = 0; geneIndex < individual.
        getChromosomeLength(); geneIndex++) {
                // Skip mutation if this is an elite individual
                if (populationIndex >= this.elitismCount) {
                        // Does this gene need mutation?
                        if (this.mutationRate > Math.random()) {
                                // Get new gene
                                int newGene = 1;
                                if (individual.getGene(geneIndex) == 1) {
                                        newGene = 0;
                                }
                                // Mutate gene
                                individual.setGene(geneIndex, newGene);
                        }
                }
        }
        // Add individual to population
        newPopulation.setIndividual(populationIndex, individual);
```

```
        }

        // Return mutated population
        return newPopulation;
    }
```

mutatePopulation( )方法开始为变异的个体创建一个新的空种群，然后开始遍历当前种群。然后，遍历每个个体的染色体，基于变异率，考虑每个基因是否进行位翻转变异。个体的整个染色体遍历完成后，该个体就被加入新的变异种群。所有的个体都通过变异过程后，返回该变异种群。

现在，我们可以完成进化循环的最后一步，将变异功能加到 main 方法。完成的 main 方法如下。与前面相比有两处不同：首先，我们在“Apply mutation”注释的下面添加了 mutatePopulation()调用。此外，既然已经明白了精英主义的工作原理，我们在 new GeneticAlgorithm 构造方法中，将 elitismCount 参数从 0 改为 2。

```
package chapter2;
public class AllOnesGA {

    public static void main(String[] args) {
        // Create GA object
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001, 0.95, 2);

        // Initialize population
        Population population = ga.initPopulation(50);

        // Evaluate population
        ga.evalPopulation(population);
```

```
        // Keep track of current generation
        int generation = 1;

        while (ga.isTerminationConditionMet(population) == false) {
            // Print fittest individual from population
            System.out.println("Best solution: " + population.
            getFittest(0).toString());

            // Apply crossover
            population = ga.crossoverPopulation(population);

            // Apply mutation
            population = ga.mutatePopulation(population);

            // Evaluate population
            ga.evalPopulation(population);

            // Increment the current generation
            generation++;
        }

        System.out.println("Found solution in " + generation + "
        generations");
        System.out.println("Best solution: " + population.
        getFittest(0).toString());
    }
}
```

2.8.3 执行

现在，你已经完成了第一个遗传算法。Individual 和 Population 类在本章前面已经完全打印，你的这些类应该看起来就像上面一样。最后的 AllOnesGA

执行类（引导和运行算法的类）直接在上面提供。

GeneticAlgorithm 类相当长，你一块一块地创建了它，所以此时，请检查是否已经实现了以下属性和方法。为了节省篇幅，这里省略了所有的注释和方法体（我只是显示了类的折叠视图），但要确保你的类有这些方法，像前面介绍的一样实现。

```
package chapter2;

public class GeneticAlgorithm {
        private int populationSize;
        private double mutationRate;
        private double crossoverRate;
        private int elitismCount;

        public GeneticAlgorithm(int populationSize, double mutationRate, double
        crossoverRate, int elitismCount) { }
        public Population initPopulation(int chromosomeLength) { }
        public double calcFitness(Individual individual) { }
        public void evalPopulation(Population population) { }
        public boolean isTerminationConditionMet(Population population) { }
        public Individual selectParent(Population population) { }
        public Population crossoverPopulation(Population population) { }
        public Population mutatePopulation(Population population) { }
}
```

如果你使用 Eclipse IDE，现在可以运行该算法，打开 AllOnesGA 文件，然后点击“Run”按钮，该按钮通常在 IDE 的顶部菜单中。

在运行时，该算法将信息打印到控制台，点击 Run 时，控制台会自动出现在 Eclipse 中。因为每一个遗传算法的随机性，每次运行看起来有点不同，但输出看起来可能如下所示：

```
Best solution: 1100111010011011111101011100100110011111100111111111
Best solution: 1100111010011011111101011100100110011111100111111111
Best solution: 1100111010011011111101011100100110011111100111111111
[ ... Lots of lines omitted here ... ]
Best solution: 1111111111111111111111111111101111111111111111111111
Best solution: 1111111111111111111111111111101111111111111111111111
Found solution in 113 generations
Best solution: 1111111111111111111111111111111111111111111111111111
```

此时，你应该尝试给 GeneticAlgorithm 构造方法提供不同的参数：populationSize、mutationRate、crossoverRate 和 elitismCount。不要忘记，统计决定了遗传算法的表现，所以不能通过运行一次来评价一个算法或一种设置的表现：每种不同的设置至少要尝试 10 次，才能判断其表现。

2.9 小结

在本章中，你已经学会了实现遗传算法的基本知识。本章开头的伪代码提供了一个通用的概念模型，针对本书其余部分所有要实现的遗传算法：每个遗传算法将初始化并评估种群，然后进入一个循环，进行交叉、变异和再评估。仅当终止条件满足时，才退出循环。

在本章中，你建立了遗传算法的支持组件，尤其是 Individual 和 Population 类，在后面的章节中基本上会复用它们。然后你专门建立了 GeneticAlgorithm 类，具体解决“全一”问题，并成功地运行了它。

你还了解了以下内容：虽然每个遗传算法在概念上和结构上相似，但不同的问题领域需要评估技术（即适应度评分、交叉技术和变异技术）的不同实现。

本书其余的部分将通过示例问题，探讨这些不同的技术。在后面的章节中，你将复用 Population 和 Individual 类，只需轻微的修改。然而，后面每章将大量修改 GeneticAlgorithm 类，因为这个类是交叉、变异、终止条件和适应度评估发生的地方。

2.10 练习

1．运行遗传算法几次，观察进化过程的随机性。它通常需要多少代来找到这个问题的一个解？

2．扩大和减小种群规模。减小种群规模如何影响算法的速度？它是否也影响找到一个解需要的世代数？扩大种群规模如何影响算法的速度？它如何影响找到一个解需要的世代数？

3．将变异率设置为 0。这将如何影响遗传算法寻找解的能力？使用高变异率，如何影响算法？

4．使用低交叉率。低交叉率下，算法表现如何？

5．尝试用较短及较长的染色体，减少和增加问题的复杂性。在处理更短或更长的染色体时，不同的参数是否工作得更好？

6．启用或不启用精英，比较遗传算法的表现。

7．采用较高的精英主义值运行测试。这将如何影响搜索表现？

第3章

机器人控制器

3.1 简介

在本章中，我们将利用前一章获得的知识，用遗传算法来解决现实世界的问题。我们要解决的现实世界问题，是设计机器人控制器。

遗传算法常常应用于机器人，作为设计复杂的机器人控制器的方法，让机器人能执行复杂的任务和行为，不需要手工编码复杂的机器人控制器。想象一下，你建立了一个可以在仓库中送货的机器人。你安装了传感器，这让机器人看到了当地的环境，你给它装了车轮，因此它可以基于传感器的输入来行驶。现在的问题是，如何将传感器数据与马达的动作联系起来，让机器人能在仓库中穿行。

将遗传算法，或更一般来说，将达尔文进化论思想应用于机器人的人工智能领域，被称为进化机器人。但是，这不是针对此问题的唯一的自下而上的方法。神经网络也常常成功地将机器人传感器映射到输出，它利用强化学习算法来指导学习过程。

通常，遗传算法会评估一大群个体，确定最佳的一些个体作为下一代。评

估个体是通过运行适应度函数完成的，该函数基于某个预定义的判据来测量个体的表现。但是，将遗传算法及其适应度函数应用于物理机器人时，产生了一个新的挑战：针对很大的种群，物理评估每个机器人控制器是不可行的。这是因为物理上测试每个机器人控制器很难，并且这样做所需的时间很长。出于这个原因，机器人控制器通常的评估方法是，将它们应用于真实物理机器人和环境的模拟模型。这样就能在软件上快速评估每个控制器，稍后可以将软件应用于物理机器人。在本章中，我们将利用二进制遗传算法的知识，来设计机器人控制器，并开始将它应用于虚拟环境中的虚拟机器人。

3.2 问题

我们要解决的问题是设计一个机器人控制器，利用机器人的传感器来导航机器人，成功通过一个迷宫。该机器人可以执行 4 种动作：向前一步、左转、右转，或者很少情况下，什么也不做。机器人有 6 个传感器：3 个在前面、1 个在左边、1 个在右边、1 个在后面，如图 3-1 所示。

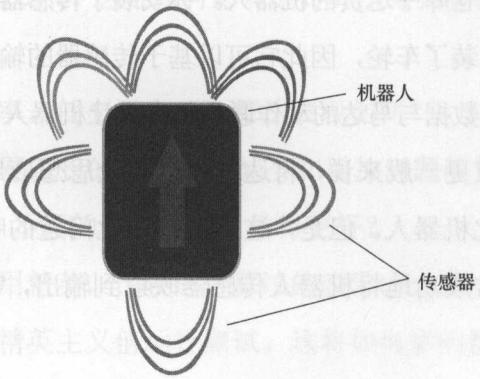


图 3-1 机器人和传感器

我们要探索的迷宫由墙构成，机器人不能穿墙，会有一条画出的路线，如图 3-2 所示，我们希望机器人遵循。请记住，本章的目的不是训练一个机器人来解决迷宫。我们的目的是，自动编程带有 6 个传感器的机器人控制器，让它不会撞墙，我们只是用迷宫作为一个复杂的环境，来测试机器人控制器。

每当机器人的传感器检测到墙靠近传感器时，就会激活。例如，如果它检测到机器人前面是墙，机器人的前传感器将激活。

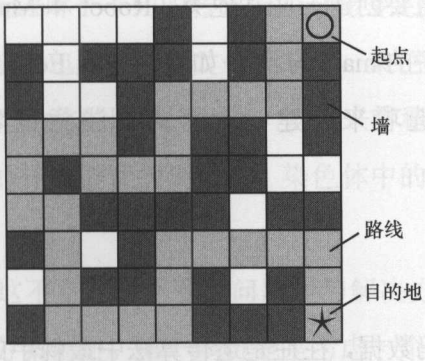


图 3-2　我们希望机器人遵循的路线

3.3 实现

3.3.1 开始之前

本章基于第 2 章开发的代码。开始之前，先创建一个新的 Eclipse 或 NetBeans 项目，或在针对本书的项目中创建一个新包，名为“chapter3”。

从第 2 章复制 Individual、Population 和 GeneticAlgorithm 类，并将它们导入到 chapter3 中。确保更新每个类文件顶部的包名，应该在它们的顶部都加上“package chapter3”。

在本章中，你完全不必修改 Individual and Population 类，只要将包名改为“chapter3”。

但是，你要修改 GeneticAlgorithm 类的几个方法。此时，你应该完全删除以下 5 个方法：calcFitness、evalPopulation、isTerminationConditionMet、selectParent 和 crossoverPopulation。在本章中，你会重写这 5 个方法，现在删除它们可以确保不会因为不小心而复用第 2 章的实现。

在本章中，你还需要创建一些其他类（Robot 和 Maze，还有执行的 Robot Controller 类，包含程序的 main 方法）。如果你使用 Eclipse，很容易通过 File ➤ New ➤ Class 菜单选项来创建一个新类。注意包名称字段，并确保是“chapter3”。

3.3.2 编码

用正确的方式编码数据，往往是遗传算法中最棘手的问题。让我们先从定义问题开始：我们需要机器人控制器的完整指令集的一种二进制表示，针对所有可能的输入组合。

如前所述，我们的机器人有 4 个动作：什么也不做，往前走一步，左转以及右转。这些可以以二进制为表示：

- “00”：什么都不做；
- “01”：前进；
- “10”：左转；
- “11”：右转。

我们还有 6 个不同的开/关传感器，给出 26（64）种可能的传感器输入组合。如果每个操作需要 2 位来编码，就可以用 128 位表示控制器的任何可能输入。换言之，我们有 64 个不同传感器状态，机器人可以处于其中一种，我们

的控制器需要为每种情况确定一个动作。因为一个动作需要两个位，我们的控制器需要存储 64×2 =128 位。

由于遗传算法染色体作为数组最容易操作，我们的染色体将采用长度 128 位的数组。在这种情况下，利用我们的变异和交叉方法，你不必担心它们正在修改哪个特定的指令，它们只是去操作遗传密码。但在最后，我们必须解开编码数据，然后将它用于机器人控制器。

鉴于 128 位是表示 64 个不同的传感器组合指令的要求，我们应该如何构建实际的染色体，以便打包和解包呢？也就是说，染色体的每个部分对应于哪种传感器输入的组合？动作的顺序是什么？对于“前和右前传感器被激活”的情形，我们在染色体的什么位置找到动作？染色体中的位表示输出，但输入如何表示？

对许多人来说，这不是一个直观的问题（和解决方案），所以让我们以一些小步骤，朝着解决方案前进。第一步可能是考虑一个简单的、人类可读的输入输出列表：

```
传感器 #1 (前): on
传感器 #2 (左前): off
传感器 #3 (右前): on
传感器 #4 (左): off
传感器 #5 (右): off
传感器 #6 (后): off
指令：左转（上面定义的动作“10”）
```

还有 63 种情况表示所有可能的组合，这种格式不方便。很明显，这样的枚举不适合。我们再前进一小步，缩写所有传感器，将“on”和“off”翻译成 1 和 0：

```
#1: 1
#2: 0
#3: 1
#4: 0
#5: 0
#6: 0

指令: 10
```

我们已经取得了一些进展，但这仍然没有将 64 个指令打包成一个 128 位的数组。下一步是取得 6 个传感器的值（输入），并进一步编码。将它们从右到左排队，并扔掉输出中的单词“指令”：

```
#6:0, #5:0, #4:0, #3:1, #2:0, #1:1 => 10
```

现在，扔掉传感器编号：

```
000101 => 10
```

如果现在将传感器值的比特串转换为十进制，就得到：

```
5 => 10
```

现在，我们有所发现。左边的“5”表示传感器输入，右边的“10”表示当面对这些输入时，机器人应该做什么（输出）。因为我们从传感器输入的二进制表示推导而来，所以只有一种传感器组合能够给出数字 5。

我们可以用数字 5 作为染色体中的位置，表示传感器输入的一种组合。如果我们手工建立这个染色体，就知道“10”（左转）是“5”（前和右前传感器检测到墙）的正确反应，我们将“1”和“0”放在染色体的第 11 和 12 个位置（每个动作需要 2 位，我们从 0 开始数位置），如下所示：

```
xx xx xx xx xx 10 xx xx xx xx (... 54 more pairs...)
```

在上面的假染色体中，第一对（位置 0）表示传感器输入组合为 0 时的动作：全都是 off。第二对（位置 1）表示传感器输入组合为 1 时的动作：只有前传感器检测到墙。第三对（位置 2），表示只有左前传感器触发。第四对（位置 3），表示前部和左前传感器都被激活。依此类推，直到最后一对（位置 63），表示所有传感器被触发。

这种编码方案的另一种展现如图 3-3 所示。最左边的“传感器”列表示传感器的位域，将位域转换成十进制后，就对应到染色体上的一个位置。将传感器位域转换成十进制后，就可以将所需的操作放在染色体的对应位置。

这种编码方案一开始会让人觉得不明显（并且染色体不是人类可读的），但它有几个有用的特点。首先，染色体可以作为位数组来操作，而不是一个复杂的树状结构或散列映射，这使得交叉、变异和其他操作要容易得多。其次，每个 128 位的值都是一个有效的解（虽然不一定是好的），本章稍后会继续讨论这一点。

图 3-3 描述了典型的染色体如何将机器人的传感器值映射到动作。

传感器						动作	染色体
B	R	L	FR	FL	F		
0	0	0	0	0	0	11	**11**001001101001...11
0	0	0	0	0	1	00	1**10**01001101001...11
0	0	0	0	1	0	10	11001**10**01101001...11
0	0	0	0	1	1	01	11001001**01**01001...11
0	0	0	1	0	0	10	11001001**10**1001...11
0	0	0	1	0	1	10	11001001101**0**01...11
0	0	0	1	1	0	01	11001001101**01**...11
1	1	1	1	1	1	11	11001001101001...**11**

图 3-3　将传感器值对应到动作

3.3.3 初始化

在这个实现中，首先需要创建和初始化一个迷宫，在其中运行机器人。要做到这一点，创建下面的 Maze 类来管理迷宫。这可以用下面的代码来完成。在 Eclipse 中，选择 File ➤ New ➤ Class，创建一个新类，并确保使用正确的包名，特别是如果你从第 2 章复制了文件。

```
package chapter3;

import java.util.ArrayList;

public class Maze {
    private final int maze[][];
    private int startPosition[] = { -1, -1 };

    public Maze(int maze[][]) {
        this.maze = maze;
    }

    public int[] getStartPosition() {
        // Check if we've already found start position
        if (this.startPosition[0] != -1 && this.startPosition[1] != -1) {
            return this.startPosition;
        }

        // Default return value
        int startPosition[] = { 0, 0 };
```

```
        // Loop over rows
        for (int rowIndex = 0; rowIndex < this.maze.length;
        rowIndex++) {
                // Loop over columns
                for (int colIndex = 0; colIndex < this.maze[rowIndex].
                length; colIndex++) {
                        // 2 is the type for start position
                        if (this.maze[rowIndex][colIndex] == 2) {
                                this.startPosition = new int[] {
                                colIndex, rowIndex };
                                return new int[] { colIndex, rowIndex };
                        }
                }
        }
        return startPosition;
}

public int getPositionValue(int x, int y) {
        if (x < 0 || y < 0 || x >= this.maze.length || y >=
        this.maze[0].length) {
                return 1;
        }
        return this.maze[y][x];
}

public boolean isWall(int x, int y) {
```

```
        return (this.getPositionValue(x, y) == 1);
    }

    public int getMaxX() {
        return this.maze[0].length - 1;
    }

    public int getMaxY() {
        return this.maze.length - 1;
    }

    public int scoreRoute(ArrayList<int[]> route) {
        int score = 0;
        boolean visited[][] = new boolean[this.getMaxY() + 1]
         [this.getMaxX() + 1];

        // Loop over route and score each move
        for (Object routeStep : route) {
            int step[] = (int[]) routeStep;
            if (this.maze[step[1]][step[0]] == 3 &&
              visited[step[1]][step[0]] == false) {
                // Increase score for correct move
                score++;
                // Remove reward
                visited[step[1]][step[0]] = true;
            }
        }
        return score;
```

```
        }
    }
```

这段代码包含一个构造方法，它从一个二维 int 数组创建一个新迷宫，以及一些公有方法，取得起始位置，检查位置的值，并对通过迷宫路线评分。

scoreRoute 方法是 Maze 类中最重要的方法，它评估由机器人采取的路径，基于它踩到正确地砖的数目，返回适应度得分。scoreRoute 方法返回的得分，稍后将作为 GeneticAlgorithm 类的 calcFitness 方法中个体的适应度评分。

既然有了迷宫抽象，我们就可以创建执行类（实际执行算法的类），并初始化图 3-2 所示的迷宫。创建另一个新类，名为 RobotController，并创建 main 方法，作为程序的入口。

```
package chapter3;

public class RobotController {

        public static int maxGenerations = 1000;

        public static void main(String[] args) {

                /**
                 * 0 = Empty
                 * 1 = Wall
                 * 2 = Starting position
                 * 3 = Route
                 * 4 = Goal position
                 */
```

```
        Maze maze = new Maze(new int[][] {
                { 0, 0, 0, 0, 1, 0, 1, 3, 2 },
                { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
                { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
                { 3, 3, 3, 1, 3, 1, 1, 0, 1 },
                { 3, 1, 3, 3, 3, 1, 1, 0, 0 },
                { 3, 3, 1, 1, 1, 1, 0, 1, 1 },
                { 1, 3, 0, 1, 3, 3, 3, 3, 3 },
                { 0, 3, 1, 1, 3, 1, 0, 1, 3 },
                { 1, 3, 3, 3, 3, 1, 1, 1, 4 }
        });
        /**
         * We'll implement the genetic algorithm pseudocode
         * from chapter 2 here....
         */
    }
}
```

我们创建的迷宫对象使用整数来表示不同的地形类型：1 表示墙，2 是起始位置，3 记录了穿过迷宫的最佳途径，4 是目标位置，0 是空位置，机器人可以走到空位置，但它们并不在通往目标位置的路线上。

接下来，类似我们以前的实现，需要初始化随机个体的种群。每个个体都应该有一条长度为 128 的染色体。如前所述，128 位让我们将所有 64 种输入映射到动作。因为不可能为这一问题创造无效的染色体，所以可以像以前一样采用同样的随机初始化：回忆一下，这种随机初始化发生在 Individual 类的构造方法，我们从第 2 章复制过来，未作改动。遇到不同情况时，用这种方式初始化的机器人只会采取随机动作。通过若干世代的进化，我们希望能够优化这种行为。

在 main 方法中勾勒出来自第 2 章的熟悉的遗传算法伪代码之前，我们应该对从第 2 章复制来 GeneticAlgorithm 类进行一处修改。我们要添加一个名为 tournamentSize 的属性（本章稍后我们将深入讨论），加入 GeneticAlgorithm 类和构造方法。

修改 GeneticAlgorithm 类的开始部分，使其如下所示：

```
package chapter3;
public class GeneticAlgorithm {

        /**
         * See chapter2/GeneticAlgorithm for a description of these
           properties.
         */
        private int populationSize;
        private double mutationRate;
        private double crossoverRate;
        private int elitismCount;

        /**
         * A new property we've introduced is the size of the population
           used for
         * tournament selection in crossover.
         */
       protected int tournamentSize;

        public GeneticAlgorithm(int populationSize, double mutationRate,
        double crossoverRate, int elitismCount,
                int tournamentSize) {
```

```
        this.populationSize = populationSize;
        this.mutationRate = mutationRate;
        this.crossoverRate = crossoverRate;
        this.elitismCount = elitismCount;
        this.tournamentSize = tournamentSize;
    }

    /**
     * We're not going to show the rest of the class here,
     * but methods like initPopulation, mutatePopulation,
     * and evaluatePopulation should appear below.
     */
}
```

我们进行了 3 处简单的改动：首先，在类属性中增加了 protected int tournamentSize。其次，添加了 int tournamentSize 作为构造方法的第 5 个参数。最后，在构造方法中添加了 this.tournamentSize= tournamentSize 赋值。

处理好 tournamentSize 属性后，我们可以继续前进，勾勒出来自第 2 章的伪代码。像往常一样，这段代码会在执行类的 main 方法中，这里我们将该类命名为 RobotController。

下面的代码不会做任何事情，当然，我们还没有实现任何需要的方法，现在只是用 TODO 注释代替这一切。但以这种方式标记出 main 方法的存根，有助于强化遗传算法的概念执行模型，也有助于我们记住那些还需要实现的方法；在这个类中，有七处代码还有待实现。

更新 RobotController 类，像下面这样。迷宫定义和以前一样，但下面的内容都是新加入该文件的。

```
package chapter3;

public class RobotController {

    public static int maxGenerations = 1000;

    public static void main(String[] args) {

        Maze maze = new Maze(new int[][] {
                { 0, 0, 0, 0, 1, 0, 1, 3, 2 },
                { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
                { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
                { 3, 3, 3, 1, 3, 1, 1, 0, 1 },
                { 3, 1, 3, 3, 3, 1, 1, 0, 0 },
                { 3, 3, 1, 1, 1, 1, 0, 1, 1 },
                { 1, 3, 0, 1, 3, 3, 3, 3, 3 },
                { 0, 3, 1, 1, 3, 1, 0, 1, 3 },
                { 1, 3, 3, 3, 3, 1, 1, 1, 4 }
        });

        // Create genetic algorithm
        GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.05,
        0.9, 2, 10);
        Population population = ga.initPopulation(128);

        // TODO: Evaluate population

        int generation = 1;
```

```
            // Start evolution loop
            while (/* TODO */ false) {
                // TODO: Print fittest individual from population

                // TODO: Apply crossover

                // TODO: Apply mutation

                // TODO: Evaluate population

                // Increment the current generation
                generation++;
            }
            // TODO: Print results
        }
    }
```

RobotController 类和第 2 章的 AllOnesGA 类只有一点不同。AllOnesGA 类没有 maxGenerations 属性，因为我们清楚地知道目标的适应度评分是什么。但在这个例子中，我们将学习用另一种方式来结束进化循环。AllOnesGA 类也不需要一个 Maze 类，但在真实的遗传算法问题中，经常发现 Maze 这样的支持类。此外，这个版本的 GeneticAlgorithm 类需要 5 个参数，而不是 4 个，因为我们希望在本章中引入锦标赛选择（tournament selection）的新概念。最后，在这个例子中，染色体的长度是 128，而不是第 2 章中的 50。在最后一章中（第 6 章），染色体长度是任意的，但在这个例子中，染色体长度是有意义的，由前面讨论过的编码方法决定。

3.3.4 评估

在评估阶段，我们需要定义一个适应度函数，它能评估每个机器人控制器。针对路线上的每次正确移动，我们可以增加个体的适应度，从而做到这一点。回想一下，我们在前面创建的 Maze 类有一个 scoreRoute 方法，执行此评估。然而，路线本身来自自主控制下的机器人。所以，在我们给 Maze 类评估的路线之前，先要创建一个可以遵循指令的机器人，它通过执行这些指令而生成路线。

创建一个 Robot 类来管理机器人的功能。在 Eclipse 中，你可以通过选择菜单选项 File ➤ New ➤ Class 来创建一个新类。确保使用正确的包名。将下面的代码添加到该文件中：

```
package chapter3;
import java.util.ArrayList;

/**
 * A robot abstraction. Give it a maze and an instruction set, and it will
 * attempt to navigate to the finish.
 *
 * @author bkanber
 *
 */
public class Robot {
    private enum Direction {NORTH, EAST, SOUTH, WEST};

    private int xPosition;
    private int yPosition;
```

```
    private Direction heading;
    int maxMoves;
    int moves;
    private int sensorVal;
    private final int sensorActions[];
    private Maze maze;
    private ArrayList<int[]> route;
    /**
     * Initalize a robot with controller
     *
     * @param sensorActions The string to map the sensor value to actions
     * @param maze The maze the robot will use
     * @param maxMoves The maximum number of moves the robot can make
     */
    public Robot(int[] sensorActions, Maze maze, int maxMoves){
        this.sensorActions = this.calcSensorActions(sensorActions);
        this.maze = maze;
        int startPos[] = this.maze.getStartPosition();
        this.xPosition = startPos[0];
        this.yPosition = startPos[1];
        this.sensorVal = -1;
        this.heading = Direction.EAST;
        this.maxMoves = maxMoves;
        this.moves = 0;
        this.route = new ArrayList<int[]>();
        this.route.add(startPos);
    }
```

```
/**
 * Runs the robot's actions based on sensor inputs
 */
public void run(){
    while(true){
        this.moves++;

        // Break if the robot stops moving
        if (this.getNextAction() == 0) {
            return;
        }

        // Break if we reach the goal
        if (this.maze.getPositionValue(this.xPosition,
        this.yPosition) == 4) {
            return;
        }

        // Break if we reach a maximum number of moves
        if (this.moves > this.maxMoves) {
            return;
        }

        // Run action
        this.makeNextAction();
    }
}
```

```
/**
 * Map robot's sensor data to actions from binary string
 *
 * @param sensorActionsStr Binary GA chromosome
 * @return int[] An array to map sensor value to an action
 */
private int[] calcSensorActions(int[] sensorActionsStr){
    // How many actions are there?
    int numActions = (int) sensorActionsStr.length / 2;
    int sensorActions[] = new int[numActions];

    // Loop through actions
    for (int sensorValue = 0; sensorValue < numActions; sensorValue++){
        // Get sensor action
        int sensorAction = 0;
        if (sensorActionsStr[sensorValue*2] == 1){
            sensorAction += 2;
        }
        if (sensorActionsStr[(sensorValue*2)+1] == 1){
            sensorAction += 1;
        }

        // Add to sensor-action map
        sensorActions[sensorValue] = sensorAction;
    }

    return sensorActions;
}
```

```
/**
 * Runs the next action
 */
public void makeNextAction(){
    // If move forward
    if (this.getNextAction() == 1) {
        int currentX = this.xPosition;
        int currentY = this.yPosition;

        // Move depending on current direction
        if (Direction.NORTH == this.heading) {
            this.yPosition += -1;
            if (this.yPosition < 0) {
                this.yPosition = 0;
            }
        }

        else if (Direction.EAST == this.heading) {
            this.xPosition += 1;
            if (this.xPosition > this.maze.getMaxX()) {
                this.xPosition = this.maze.getMaxX();
            }
        }
        else if (Direction.SOUTH == this.heading) {
            this.yPosition += 1;
            if (this.yPosition > this.maze.getMaxY()) {
                this.yPosition = this.maze.getMaxY();
```

```
                }
            }
            else if (Direction.WEST == this.heading) {
                this.xPosition += -1;
                if (this.xPosition < 0) {
                    this.xPosition = 0;
                }
            }
            // We can't move here
            if (this.maze.isWall(this.xPosition, this.yPosition) == true) {
                this.xPosition = currentX;
                this.yPosition = currentY;
            }
            else {
                if(currentX != this.xPosition || currentY
                != this.yPosition) {
                    this.route.add(this.getPosition());
                }
            }
        }
        // Move clockwise
        else if(this.getNextAction() == 2) {
            if (Direction.NORTH == this.heading) {
                this.heading = Direction.EAST;
            }
            else if (Direction.EAST == this.heading) {
                this.heading = Direction.SOUTH;
```

```
            }
            else if (Direction.SOUTH == this.heading) {
                this.heading = Direction.WEST;
            }
            else if (Direction.WEST == this.heading) {
                this.heading = Direction.NORTH;
            }
        }

        // Move anti-clockwise
        else if(this.getNextAction() == 3) {
            if (Direction.NORTH == this.heading) {
                this.heading = Direction.WEST;
            }
            else if (Direction.EAST == this.heading) {
                this.heading = Direction.NORTH;
            }
            else if (Direction.SOUTH == this.heading) {
                this.heading = Direction.EAST;
            }
            else if (Direction.WEST == this.heading) {
                this.heading = Direction.SOUTH;
            }
        }

        // Reset sensor value
        this.sensorVal = -1;
    }
```

```
/**
 * Get next action depending on sensor mapping
 *
 * @return int Next action
 */
public int getNextAction() {
    return this.sensorActions[this.getSensorValue()];
}

/**
 * Get sensor value
 *
 * @return int Next sensor value
 */
public int getSensorValue(){
    // If sensor value has already been calculated
    if (this.sensorVal > -1) {
        return this.sensorVal;
    }

        boolean frontSensor, frontLeftSensor, frontRightSensor,
        leftSensor, rightSensor, backSensor;
        frontSensor = frontLeftSensor = frontRightSensor =
        leftSensor = rightSensor = backSensor = false;
    // Find which sensors have been activated
    if (this.getHeading() == Direction.NORTH) {
        frontSensor = this.maze.isWall(this.xPosition,
        this.yPosition-1);
```

```
            frontLeftSensor = this.maze.isWall(this.xPosition-1,
            this.yPosition-1);
            frontRightSensor = this.maze.isWall(this.xPosition+1,
            this.yPosition-1);
            leftSensor = this.maze.isWall(this.xPosition-1,
            this.yPosition);
            rightSensor = this.maze.isWall(this.xPosition+1,
            this.yPosition);
            backSensor = this.maze.isWall(this.xPosition,
            this.yPosition+1);
        }
        else if (this.getHeading() == Direction.EAST) {
            frontSensor = this.maze.isWall(this.xPosition+1,
            this.yPosition);
            frontLeftSensor = this.maze.isWall(this.xPosition+1,
            this.yPosition-1);
            frontRightSensor = this.maze.isWall(this.xPosition+1,
            this.yPosition+1);
            leftSensor = this.maze.isWall(this.xPosition,
            this.yPosition-1);
            rightSensor = this.maze.isWall(this.xPosition,
            this.yPosition+1);
            backSensor = this.maze.isWall(this.xPosition-1,
            this.yPosition);
        }
        else if (this.getHeading() == Direction.SOUTH) {
            frontSensor = this.maze.isWall(this.xPosition,
            this.yPosition+1);
```

```
                frontLeftSensor = this.maze.isWall(this.xPosition+1,
                this.yPosition+1);
                frontRightSensor = this.maze.isWall(this.xPosition-1,
                this.yPosition+1);
                leftSensor = this.maze.isWall(this.xPosition+1,
                this.yPosition);
                rightSensor = this.maze.isWall(this.xPosition-1,
                this.yPosition);
                backSensor = this.maze.isWall(this.xPosition,
                this.yPosition-1);
            }
            else {
                frontSensor = this.maze.isWall(this.xPosition-1,
                this.yPosition);
                frontLeftSensor = this.maze.isWall(this.xPosition-1,
                this.yPosition+1);
                frontRightSensor = this.maze.isWall(this.xPosition-1,
                this.yPosition-1);
                leftSensor = this.maze.isWall(this.xPosition,
                this.yPosition+1);
                rightSensor = this.maze.isWall(this.xPosition,
                this.yPosition-1);
                backSensor = this.maze.isWall(this.xPosition+1,
                this.yPosition);
            }

            // Calculate sensor value
            int sensorVal = 0;
```

```
        if (frontSensor == true) {
            sensorVal += 1;
        }
        if (frontLeftSensor == true) {
            sensorVal += 2;
        }
        if (frontRightSensor == true) {
            sensorVal += 4;
        }
        if (leftSensor == true) {
            sensorVal += 8;
        }
        if (rightSensor == true) {
            sensorVal += 16;
        }
        if (backSensor == true) {
            sensorVal += 32;
        }

        this.sensorVal = sensorVal;

        return sensorVal;
    }

    /**
     * Get robot's position
     *
```

```
     * @return int[] Array with robot's position
     */
    public int[] getPosition(){
        return new int[]{this.xPosition, this.yPosition};
    }

    /**
     * Get robot's heading
     *
     * @return Direction Robot's heading
     */
    private Direction getHeading(){
        return this.heading;
    }

    /**
     * Returns robot's complete route around the maze
     *
     * @return ArrayList<int> Robot's route
     */
    public ArrayList<int[]> getRoute(){
        return this.route;
    }

    /**
     * Returns route in printable format
     *
     * @return String Robot's route
```

```
     */
    public String printRoute(){
        String route = "";

        for (Object routeStep : this.route) {
            int step[] = (int[]) routeStep;
            route += "{" + step[0] + "," + step[1] + "}";
        }
        return route;
    }
}
```

这个类包含了构造方法来创建一个新 Robot。它也包含一些方法，来读取机器人的传感器，取得机器人朝向，让机器人在迷宫中四处移动。这个 Robot 类是我们模拟一个简单机器人的方法，这样就不必对 100 个实际机器人构成的种群进行 1000 代的进化。在这样的优化问题中，我们常常发现像 Maze 和 Robot 这样的类，先用软件来模拟，再在产品硬件上优化结果，这是有成本效益的。

回想一下，技术上是 Maze 类来评估路线的适应度。但是，我们仍然需要在 GeneticAlgorithm 类中实现 calcFitness 方法。calcFitness 方法不是直接计算适应度评分，而是负责用 Individual 的染色体（即传感器控制器指令集）创建一个新的 Robot，针对 Maze 评估它，从而将 Individual、Robot 和 Maze 类联系在一起。

在 GeneticAlgorithm 类中编写下面的 calcFitness 函数。像往常一样，该方法可以放在类的任何位置。

```
public double calcFitness(Individual individual, Maze maze) {
    int[] chromosome = individual.getChromosome();
    Robot robot = new Robot(chromosome, maze, 100);
```

```
        robot.run();
        int fitness = maze.scoreRoute(robot.getRoute());
        individual.setFitness(fitness);
        return fitness;
    }
```

这里，calcFitness 方法接受两个参数，individual 和 maze，用于建立新的机器人，并让它在迷宫中行走。然后对机器人的路线评分，并保存为个体的适应度。

这段代码创建了一个机器人，把它放在迷宫中，并用进化的控制器对其进行测试。Robot 构造方法的最后一个参数是允许机器人移动的最大步数。这将防止它陷入死胡同，或者永不停止地绕圈。然后，我们可以简单地取得机器人路线的评分，利用 Maze 的 scoreRoute 方法，将它作为适应度返回。

有了有效的 calcFitness 方法，我们就可以创建一个 evalPopulation 方法。回想一下第 2 章的 evalPopulation 方法，它只是遍历种群中的每个个体，并针对该个体调用 calcFitness，在遍历时对总体种群的适应度求和。事实上，本章的 evalPopulation 几乎和第 2 章的相同：但在这个例子中，我们还需要将迷宫对象传递给 calcFitness 方法，所以需要稍作修改。

将下面的方法添加到 GeneticAlgorithm 类中，可以放在任意位置：

```
public void evalPopulation(Population population, Maze maze) {
        double populationFitness = 0;

        for (Individual individual : population.getIndividuals()) {
                populationFitness += this.calcFitness(individual, maze);
        }
```

```
        population.setPopulationFitness(populationFitness);
    }
```

这个版本与第 2 章的版本之间的唯一区别是将 Maze maze 作为第 2 个参数，并且将 maze 作为第 2 个参数传给 calcFitness。

此时，你可以解决 RobotController 的 main 方法中的两处 TODO: Evaluate population 了。找到显示以下行的两个位置：

```
// TODO: Evaluate population
```

将它们替换为：

```
// Evaluate population
ga.evalPopulation(population, maze);
```

和第 2 章中不同，该方法需要传入迷宫对象作为第 2 个参数。现在，RobotController 的 main 方法中应该只剩下 5 个“TODO”注释了。我们将在下一节中很快解决其中 3 个。这就是进步！

3.3.5 终止检查

这个实现的终止检查与前一个遗传算法中使用的略有不同。这里，我们将在达到最大世代数目后终止。

为了添加这种终止检查，先在 GeneticAlgorithm 类中加入 isTerminationConditionMet 方法，像下面这样。

```
public boolean isTerminationConditionMet(int generationsCount, int
maxGenerations) {
        return (generationsCount > maxGenerations);
    }
```

该方法只是接受当前世代计数和允许的最大世代，根据算法是否应该结束，返回 true 或 false。说实话，这很简单，我们可以在遗传算法的 while 循环中直接使用该逻辑，但是，考虑到一致性，我们总是将终止条件检查实现为 GeneticAlgorithm 类中的一个方法，即使它像上面这样，是一个简单的方法。

现在，我们可以在 RobotController 的 main 方法中添加以下代码，将终止检查应用于进化循环。我们只要将世代计数和最大世代数作为参数传入。

向 while 语句添加了终止条件，循环基本上就能工作了，所以我们也应该借此机会，打印出一些统计数据和调试信息。

下面的改动很简单：首先，更新 while 条件，使用 ga.isTerminationConditionMet。其次，为了显示进度和结果，在循环中和循环后添加对 population.getFittest 和 System.out.println 的调用。

下面是此时 RobotController 类的样子，我们刚才又解决了 3 个 TODO 部分，现在只剩下 2 个了：

```
package chapter3;

public class RobotController {

    public static int maxGenerations = 1000;

    public static void main(String[] args) {

        Maze maze = new Maze(new int[][] {
            { 0, 0, 0, 0, 1, 0, 1, 3, 2 },
            { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
            { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
```

```
        { 3, 3, 3, 1, 3, 1, 1, 0, 1 },
        { 3, 1, 3, 3, 3, 1, 1, 0, 0 },
        { 3, 3, 1, 1, 1, 1, 0, 1, 1 },
        { 1, 3, 0, 1, 3, 3, 3, 3, 3 },
        { 0, 3, 1, 1, 3, 1, 0, 1, 3 },
        { 1, 3, 3, 3, 3, 1, 1, 1, 4 }
});

// Create genetic algorithm
GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.05,
0.9, 2, 10);
Population population = ga.initPopulation(128);

// Evaluate population
ga.evalPopulation(population, maze);


int generation = 1;

// Start evolution loop
while (ga.isTerminationConditionMet(generation,
maxGenerations) == false) {
    // Print fittest individual from population
    Individual fittest = population.getFittest(0);
    System.out.println("G" + generation + " Best solution
     (" + fittest.getFitness() + "): " + fittest.
    toString());

    // TODO: Apply crossover
```

```
            // TODO: Apply mutation

            // Evaluate population
            ga.evalPopulation(population, maze);

            // Increment the current generation
            generation++;
        }

        System.out.println("Stopped after " + maxGenerations + "
        generations.");
        Individual fittest = population.getFittest(0);
        System.out.println("Best solution
         (" + fittest.getFitness() + "): " + fittest.toString());
    }
}
```

如果你现在点击 Run 按钮，就会看到算法很快循环 1000 代（没有实际的进化！），并自豪地向你提供一个非常非常糟糕的解，从统计上看，适应度很有可能是 1.0。

这不足为奇，因为我们还没实现交叉或变异！正如你在第 2 章中学到的，要推动进化，至少需要这些机制中的一个，但一般来说，两种机制都需要，以免陷入局部最优。

在上面的 main 方法中，还剩下 2 个 TODO 部分，幸运的是，我们可以非常迅速地解决其中一个。我们在第 2 章学到的变异技术（位翻转变异），也适用于这个问题。

评估一个变异或交叉算法的可行性时，首先必须考虑一些约束，即怎样才算一个有效的染色体。在这个例子中，对于这个特定问题，有效的染色体只有两个约束：它必须是二进制的，并且必须是 128 位长。只要满足这两个约束条件，所有位组合或序列都被认为是有效的。所以，我们可以复用第 2 章的简单变异方法。

实现变异很简单，和第 2 章一样。更新 TODO：Mutate population 行，将其变成下面的样子：

```
// Apply mutation
population = ga.mutatePopulation(population);
```

此时再次尝试运行该程序。结果并不惊艳，1000 代后，你可能会得到 5 或 10 的适应度评分。但有一点很明确：现在种群在进化，我们正越来越接近终点。

只剩下一个 TODO 了，这就是交叉。

3.3.6 选择方法和交叉

在以前的遗传算法，我们采用轮盘赌选择来挑选亲代，进行均匀交叉操作。回想一下，交叉是一种技术，用于结合两个亲代的遗传信息。在这个实现中，我们将使用一种新的选择方法，名为锦标赛选择（tournament selection），以及一种新的交叉方法，名为单点交叉（single point crossover）。

3.4 锦标赛选择

像轮盘赌选择一样，锦标赛选择提供了一种方法，根据个体的适应度值来选择个体。也就是说，个体的适应度越高，被选中进行交叉的机会越大。

锦标赛选择通过运行一系列的“锦标赛”来选择亲代。首先，随机从种群中选择一些个体，进入锦标赛。接下来，可以通过比较这些个体的适应度值来竞争，然后选择适应度最高的个体作为亲代。

锦标赛选择需要定义锦标赛的规模，指定从种群中选择多少个体来争夺锦标赛。像大多数参数一样，根据所选的值，在性能上会有折衷。锦标赛规模大的话，会导致考虑种群中更大的部分。这样更容易找到种群中较高得分的个体。相反，锦标赛规模小的话，会因为竞争较小，更随机地从种群中选择个体，常常导致选择较低得分的个体。锦标赛规模大可能会导致只有最好的个体被选中作为亲代，从而丧失遗传多样性。相反，锦标赛规模小则会因为减少选择压力，从而减慢算法的进展。

锦标赛选择是遗传算法中最常使用的选择方法之一。它的优势在于，这是一个相对简单的算法实现，并允许更新锦标赛的规模，从而改变选择的压力。但它也有局限性。请考虑得分最低的个体加入了一个锦标赛的情况。不论种群中其他什么个体加入这个锦标赛，它永远不会被选择，因为其他个体肯定都有较高的适应度值。这个缺点可以通过在算法中添加一个选择概率来解决。例如，如果选择概率设为 0.6，适应度最高的个体将有 60%的机会被选择。如果没有选择适应度最高的个体，就转向适应度次高的个体，如此下去，直到选中一个个体。虽然这种修改使得排名最差的个体偶尔也会被选择，但它并没有考虑个体的适应度差别。例如，如果 3 个个体被选中参加锦标赛选择，一个适应度值为 9，一个适应度值为 2，另一个适应度值为 1。在这种情况下，适应度值为 2 的个体就算适应度值是 8，也不会增加选中的可能性。这意味着，个体被选中的概率偶尔会不合理地高或低。

我们不会在锦标赛选择实现中采用选择概率，但是，它对于读者来说是极好的练习。

为了实现锦标赛选择，将下面的代码添加到 GeneticAlgorithm 类中，可以放在任意位置：

```
public Individual selectParent(Population population) {
    // Create tournament
    Population tournament = new Population(this.tournamentSize);

    // Add random individuals to the tournament
    population.shuffle();
    for (int i = 0; i < this.tournamentSize; i++) {
        Individual tournamentIndividual = population.getIndividual(i);
        tournament.setIndividual(i, tournamentIndividual);
    }

    // Return the best
    return tournament.getFittest(0);
}
```

首先，我们创建一个新的种群，存放锦标赛选择中的所有个体。接着，个体被随机添加到种群中，直到规模等于锦标赛规模参数。最后，最佳个体从比赛种群中被选中并返回。

3.5 单点交叉

单点交叉可以替代我们之前实现的均匀交叉方法。单点交叉是一种非常简单的交叉方法，在这种方法中，我们随机选择基因组中的一个位置，确定哪些

基因来自于哪个亲代。交叉位置之前的遗传信息来自于亲代 1，之后的遗传信息来自于亲代 2，如图 3-4 所示。

亲代1	1	0	0	1	1
亲代2	0	0	1	1	0
后代	1	0	1	1	0

图 3-4 单点交叉

单点交叉比较容易实现，与均匀交叉相比，它允许连续的位数组从亲代更有效地转移，这是交叉算法的宝贵特性。请考虑我们的具体问题，其中染色体是基于 6 个传感器输入的编码指令集，并且每个指令长度超过一位。

想象一下，理想的交叉情况如下：假设亲代 1 对于 32 个传感器的操作很好，亲代 2 对于最后 16 个传感器的操作很好。如果使用第 2 章的均匀交叉技术，我们得到的位全是混乱的！单个指令将在交叉中被改变和破坏，因为均匀交叉会随机选择位交换。两位的指令也许根本不会保留，因为每个指令的两位中，一位可能会被修改。但是，单点交叉让我们能抓住这一理想情况。如果交叉点就在染色体的中部，它们的后代将从亲代 1 得到不中断的 64 位（代表 32 条指令），并从亲代 2 得到很好的 16 条指令。所以，在 64 种可能状态的 48 种中，现在后代都表现得很好。这个概念是遗传算法的支柱：后代可能比亲代任何一方更强，因为它从双方继承了最优秀的品质。

但是，单点交叉并非没有局限性。单点交叉的一个限制是，亲代基因组的某些组合是根本不可能的。例如，考虑两个亲代：一个基因组是“00100”，另一个是“10001”。只通过交叉，子代“10101”是不可能的，尽管它所需的基因在两个亲代中都有。幸运的是，我们还有一种进化机制叫变异，如果交叉和变异都实施，基因组“10101”就是可能的。

单点交叉的另一个限制是，靠左侧的基因有来自亲代 1 的偏向，靠右侧的基因有来自亲代 2 的偏向。解决这个问题可以采用两点交叉，它采用两个位置，允许分区跨越亲代基因组的边缘，如图 3-5 所示。我们将两点交叉作为练习留给读者。

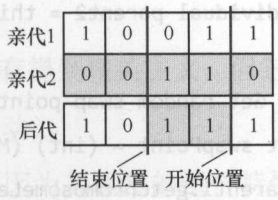


图 3-5　两点交叉

为了实现单点交叉，将下面的代码添加到 GeneticAlgorithm 类。这个 crossoverPopulation 方法依赖于前面实现的 selectParent 方法，因此使用了锦标赛选择。请注意，单点交叉没有要求必须使用锦标赛选择，你可以使用 selectParent 的任何实现，但对于这个问题，我们采用了锦标赛选择和单点交叉，因为它们都是非常常见且重要的概念，需要理解。

```
public Population crossoverPopulation(Population population) {
    // Create new population
    Population newPopulation = new Population(population.size());

    // Loop over current population by fitness
    for (int populationIndex = 0; populationIndex <
    population.size(); populationIndex++) {
        Individual parent1 = population.getFittest(populationIndex);

        // Apply crossover to this individual?
        if (this.crossoverRate > Math.random() && populationIndex >=
```

```
        this.elitismCount) {
                // Initialize offspring
                Individual offspring = new Individual
                (parent1.getChromosomeLength());

                // Find second parent
                Individual parent2 = this.selectParent(population);

                // Get random swap point
                int swapPoint = (int) (Math.random() *
                (parent1.getChromosomeLength() + 1));

                // Loop over genome
                for (int geneIndex = 0; geneIndex < parent1.
                getChromosomeLength(); geneIndex++) {
                    // Use half of parent1's genes and half of
                    parent2's genes
                    if (geneIndex < swapPoint) {
                        offspring.setGene(geneIndex,
                        parent1.getGene(geneIndex));
                    } else {
                        offspring.setGene(geneIndex,
                        parent2.getGene(geneIndex));
                    }
                }

                // Add offspring to new population
                newPopulation.setIndividual(populationIndex,
                offspring);
            } else {
                // Add individual to new population without applying
```

```
                crossover
                newPopulation.setIndividual(populationIndex, parent1);
            }
        }

        return newPopulation;
    }
```

请注意，虽然本章中没有提到精英主义，但它仍然出现在上面，在变异算法中（来自第 2 章，未作改动）。

单点交叉很受欢迎，既因为其优越的遗传特性（保持连续的基因），也因为它很容易实现。在上面的代码中，为新个体创建了一个新种群。接下来遍历亲代种群，按适应度的顺序取出个体。如果启用精英主义，精英个体被跳过，直接加入新种群，否则根据交叉率来决定是否交叉当前个体。如果个体被选择用于交叉，用锦标赛选择来挑选第二个亲代。

接下来，随机选择一个交叉点。在这个位置，我们停止使用亲代 1 的基因，开始使用亲代 2 的基因。然后，我们只要遍历染色体，先将亲代 1 的基因加给后代，再在交叉点之后切换到亲代 2 的基因。

现在，我们可以在 RobotController 的 main 方法中调用交叉了。添加代码行 population = ga.crossoverPopulation(population)，这解决了最后一个 TODO 部分，得到的 RobotController 类看起来应该如下所示：

```
package chapter3;

public class RobotController {

    public static int maxGenerations = 1000;
```

```
    public static void main(String[] args) {

        Maze maze = new Maze(new int[][] {
                { 0, 0, 0, 0, 1, 0, 1, 3, 2 },
                { 1, 0, 1, 1, 1, 0, 1, 3, 1 },
                { 1, 0, 0, 1, 3, 3, 3, 3, 1 },
                { 3, 3, 3, 1, 3, 1, 1, 0, 1 },
                { 3, 1, 3, 3, 3, 1, 1, 0, 0 },
                { 3, 3, 1, 1, 1, 1, 0, 1, 1 },
                { 1, 3, 0, 1, 3, 3, 3, 3, 3 },
                { 0, 3, 1, 1, 3, 1, 0, 1, 3 },
                { 1, 3, 3, 3, 3, 1, 1, 1, 4 }
        });

        // Create genetic algorithm
        GeneticAlgorithm ga = new GeneticAlgorithm(200, 0.05,
        0.9, 2, 10);
        Population population = ga.initPopulation(128);

        // Evaluate population
        ga.evalPopulation(population, maze);

        int generation = 1;

        // Start evolution loop
        while (ga.isTerminationConditionMet(generation,
        maxGenerations) == false) {
```

```
            // Print fittest individual from population
            Individual fittest = population.getFittest(0);
            System.out.println("G" + generation + " Best solution
            (" + fittest.getFitness() + "): " + fittest.
            toString());

            // Apply crossover
            population = ga.crossoverPopulation(population);

            // Apply mutation
            population = ga.mutatePopulation(population);

            // Evaluate population
            ga.evalPopulation(population, maze);

            // Increment the current generation
            generation++;
        }
        System.out.println("Stopped after " + maxGenerations + "
        generations.");
        Individual fittest = population.getFittest(0);
        System.out.println("Best solution
         (" + fittest.getFitness() + "): " + fittest.toString());
    }
}
```

执行

此时，GeneticAlgorithm 类应该具有以下属性和方法签名：

```
package chapter3;

public class GeneticAlgorithm {

        private int populationSize;
        private double mutationRate;
        private double crossoverRate;
        private int elitismCount;
        protected int tournamentSize;

        public GeneticAlgorithm(int populationSize, double mutationRate,
        double crossoverRate, int elitismCount, int tournamentSize) { }
        public Population initPopulation(int chromosomeLength) { }
        public double calcFitness(Individual individual, Maze maze) { }
        public void evalPopulation(Population population, Maze maze) { }
        public boolean isTerminationConditionMet(int generationsCount,
        int maxGenerations) { }
        public Individual selectParent(Population population) { }
        public Population mutatePopulation(Population population) { }
        public Population crossoverPopulation(Population population) { }
}
```

如果你的方法签名与上面不符，或者不小心遗漏了一个方法，又或者 IDE 有任何错误显示，那么你现在应该退回去并解决这些问题。

否则，点击 Run。

你应该看到 1000 个世代的进化，但愿你的算法最后得到的适应度得分是 29，这是这个特定迷宫的最大值［你可以在迷宫中的定义中数出“路线”上瓷砖的数量（以“3”表示），从而得到这个数字］。

回想一下，该算法的目的不是解决一个迷宫问题，而是为机器人传感器的控制器编程。可以假定，在执行结束时，我们可以取得获胜的染色体，将它编程写入一个物理机器人，并且非常有信心，该传感器控制器不仅会做出适当的动作，在这个迷宫中穿行，而且在任何迷宫中，都不会撞墙。不能保证该机器人会找到通过迷宫的最有效途径，因为那不是我们训练它做的事情，但最起码它不会撞墙。

尽管 64 种传感器组合对于手工编程似乎也不是太艰巨，但请考虑 3 维中同样的问题：一个无人控制、自主飞行的四轴飞行器可能有 20 个传感器，而不是 6 个。在这种情况下，你必须为 220 种传感器输入组合编程，大约 100 万条不同的指令。

3.6 小结

遗传算法可以用来设计复杂的控制器，如果人工来做，这可能很困难、很耗时。机器人控制器由适应度函数来评价，我们通常模拟机器人及其环境，不需要物理测试机器人，从而节省时间。

通过为机器人提供迷宫和优选的途径，可以用遗传算法找到一个控制器，利用机器人的传感器成功地穿行迷宫。通过在个体的染色体编码中，为每种传感器组合指定一个动作，可以做到这一点。利用交叉和变异带来的随机小变化，在选择过程的指导下，慢慢找到更好的控制器。

在遗传算法中，锦标赛选择是比较流行的选择方法之一。它的工作原理是随机从群体中挑选预定数量的个体，然后比较挑中个体的适应度值，找到最好

的。最高适应度值的个体在锦标赛中“获胜”，然后作为被选择的个体而返回。较大的锦标赛规模导致较高的选择压力，在选择最佳的锦标赛规模时，要慎重考虑。

当个体被选中后，它将进行交叉。可以使用的一种交叉方法是单点交叉。在这种交叉方法中，先在染色体上随机挑选一个单点，然后该点之前的所有遗传信息都来自于亲代A，该点之后的所有遗传信息都来自于亲代B。这导致了父母基因信息合理的随机组合，但常常用改进的两点交叉方法来代替。在两点交叉中，选择一个起点和一个终点，用它们来选择来自亲代A的遗传信息，然后剩下的遗传信息来自于亲代B。

3.7 练习

1. 添加第二个终止条件，在路线被完全探索后终止该算法。

2. 用不同的锦标赛规模运行该算法。研究它如何影响算法的表现。

3. 为锦标赛选择方法添加选择概率。用不同的概率设置来测试。研究它如何影响算法的表现。

4. 实现两点交叉，并观察它是否改进了结果。

第4章

旅行商

4.1 简介

在本章中，我们将探讨旅行商问题，以及如何用遗传算法来解决它。在此过程中，我们将研究旅行商问题的特性，以及如何利用这些特性来设计遗传算法。

旅行商问题（Traveling Salesman Problem，TSP）是一个典型的优化问题，关于它的研究可以追溯到 19 世纪。旅行商问题涉及发现经过一组城市的最有效路线，每个城市仅访问一次。

旅行商问题常常被描述为优化经过一组城市的路线，但是，旅行商问题可以应用于其他场景。例如，对于某些应用，城市的概念可以对应于客户，甚至对应于微芯片的焊接点。距离的概念也可以修改，可以考虑其他限制，例如时间。

在其最简单的形式中，城市可以表示为图上的节点，每个城市之间的距离表示为边的长度（见图 4-1）。一条“路线”或“途程”，就定义了应该使用的边，以及使用的顺序。然后通过累计路线中的边，计算出路线评分。

在20世纪，许多数学家和科学家研究了旅行商问题，然而直到今天，该问题仍然没有解决。要得到旅行商问题的最优解，唯一可以保证的方法是暴力算法。暴力算法即系统地尝试所有可能的解。然后从候选解的完整集合中找到最优解。试图用暴力算法求解旅行商问题极为困难，因为随着城市数量增加，可能解的数量呈阶乘增长。阶乘函数增长速度甚至超过了指数函数，这就是暴力解决旅行商问题如此困难的原因。例如，对于5个城市，有120个可能的解（1×2×3×4×5），对于10个城市，这个数字将增加到3628800。对于15个城市，有超过一万亿的解。对于60个城市，可能的解超过了可观测宇宙的原子数。

如果只有很少几个城市，可以用暴力算法来找到最优解，但随着城市数增加，它们就变得越来越具有挑战性。即使采用一些技术来消除回退和相同的路线，仍然很快就不可能在合理的时间内找到最佳解。

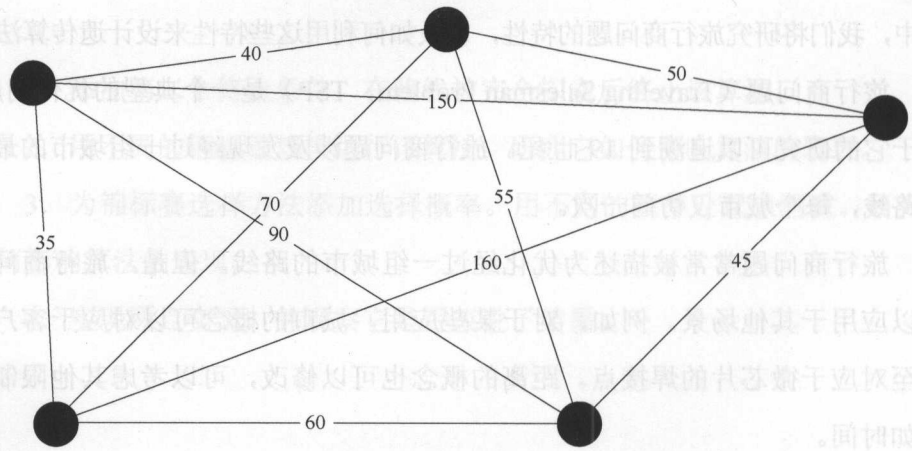


图4-1 城市和它们之间的距离

在现实中，我们知道，找到最佳解通常没有必要，因为足够好的解通常就能满足需要。有许多不同的算法，可以快速找到一些解，它们很可能属于最优

的百分之几。一种比较常用算法是最近邻算法。利用这个算法，随机挑选一个起始城市。然后，找到下一个最近的、未访问的城市，选择作为路线中的第二个城市。继续选择最近的、未访问的城市，直到所有城市都已访问，完整的路线已经找到。最近邻算法已被证明令人惊讶地有效，得到解的评分与最优解差不多。更妙的是，它可以在很短的时间内完成。这些特性使得它在许多情况下成为一个有吸引力的解决方案，甚至可能是遗传算法的替代方案。

4.2 问题

我们在这个实现中要处理的问题是典型的旅行商问题：我们需要优化通过一组城市的路线。我们可以在 2 维空间随机生成一些城市，即为每个城市设置随机的 x，y 位置。

在确定两个城市之间的距离时，就用城市间的最短距离。可以用下面的公式计算该距离：

$$\text{距离}=\sqrt{(x_a-x_b)^2+(y_a-y_b)^2}$$

通常情况下，问题比这更复杂。在这个例子中，我们假定每个城市之间存在直接的理想路线，这也称为“欧氏距离”。这通常不是典型的情况，因为可能存在各种障碍物，使得实际最短路径比欧氏距离长得多。我们还假定从市 A 到 B 市与从 B 市到 A 市所花的时间一样。同样，现实中这种情况很少见。常常会出现障碍，如单行线，影响在特定方向上行进时城市间的距离。在旅行商问题的实现中，如果城市间的距离变化取决于方向，就称为非对称旅行商问题。

4.3 实现

现在可以继续前进，利用我们的遗传算法知识来处理该问题。为该问题建立新的 Java/Eclipse 包后，我们将开始为路线编码。

4.3.1 开始之前

本章内容基于第 3 章开发的代码。开始之前，先创建一个新的 Eclipse 或 NetBeans 项目，或在针对本书的项目中创建一个新包，名为 chapter4。

从第 3 章复制 Individual、Population 和 GeneticAlgorithm 类，并将它们导入到 chapter4 中。注意确保更新每个类文件顶部的包名！它们都应该在顶部加上“package chapter4”。

打开 GeneticAlgorithm 类，删除以下方法：calcFitness、evalPopulation、crossoverPopulation 和 mutatePopulation。在本章的进程中，你会重写这些方法。

接下来，打开 Individual 类，删除签名为 public Individual(int chromosomeLength) 的构造方法。Individual 类有两个构造方法，所以要小心不要删错！要删除的构造方法是随机初始化染色体那个，你将会在本章中重写它。

除了文件顶部的包名，第 3 章的 Population 类不需要修改。

4.3.2 编码

我们在这个例子中选择的编码需要能够“按顺序”编码一个城市列表。我们可以为每个城市指定唯一的 ID，然后按照候选路线的顺序在染色体中引用它，从而做到这一点。这种类型的编码用到了基因的顺序，称为排列编码

（permutation encoding），非常适合旅行商问题。

我们要做的第一件事，就是为城市分配唯一的 ID。如果有 5 个城市要访问，我们可以简单地为它们分配 ID：1、2、3、4 和 5。然后，如果我们的遗传算法找到了路线，染色体可能将城市 ID 排序，就像：3、4、1、2 和 5。这就意味着我们从城市 3 开始，然后前往城市 4，然后城市 1，然后城市 2，然后城市 5，最后回到城市 3，完成路线。

4.3.3 初始化

开始优化路线之前，我们需要创建一些城市。正如前面提到的，我们可以挑选任意 x、y 坐标，生成随机的城市，用它们来确定城市的位置。

首先，我们需要创建一个 City 类，它可以创建并保存一个城市，并计算到另一个城市的最短距离。

```
package chapter4;
public class City {
        private int x;
        private int y;
        public City(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public double distanceFrom(City city) {
                // Give difference in x,y
                double deltaXSq = Math.pow((city.getX() - this.getX()), 2);
                double deltaYSq = Math.pow((city.getY() - this.getY()), 2);
```

```
            // Calculate shortest path
            double distance = Math.sqrt(Math.abs(deltaXSq + deltaYSq));
            return distance;
        }

        public int getX() {
            return this.x;
        }

        public int getY() {
            return this.y;
        }
    }
```

City 类有一个构造方法，接受 x 和 y 坐标，在 2 维平面上创建一个城市。该类还包含一个 distanceFrom 方法，用勾股定理计算当前城市到另一个城市的直线距离。最后，还有两个取值方法，取得城市的 x 和 y 坐标。

接下来，我们应该恢复 Individual 类的构造方法，它在 4.3.1 节中被删除。旅行商问题对染色体的约束与前两个问题不同。回想一下，机器人控制器问题的唯一约束是，染色体必须 128 位长，并且必须是二进制的。

遗憾的是，旅行商问题的情况不同，约束条件更复杂，并决定着初始化、交叉和变异中可以使用的技术。在这个例子中，染色体必须有固定的长度（与城市途程的长度一样），但额外的限制条件是每个城市必须访问一次，且只有一次，否则染色体是无效的。染色体中没有重复基因，染色体也没有漏掉的城市。

我们很容易地创建一个简单的 Individual 构造方法，没有任何随意性。只要创建一个染色体，带有每个城市的 ID：1、2、3、4、5、6 等。在本章末，随机化初始染色体留给读者作为练习。

将下面的构造方法添加到 Individual 类。你可以将它放到任何位置，但顶部附近是放置构造方法的好位置。像往常一样，注释和文档注释块在这里被省略，但请查看本书提供的 Eclipse 项目中附带的注释。

```
public Individual(int chromosomeLength) {
        // Create random individual
        int[] individual;
        individual = new int[chromosomeLength];

        for (int gene = 0; gene < chromosomeLength; gene++) {
              individual[gene] = gene;
        }

        this.chromosome = individual;
}
```

此时，我们可以创建执行类和它的 main 方法。通过 File ➤ New ➤ Class 菜单项创建一个名为 TSP 的类，放在包 chapter4 中。像第 3 章一样，我们用一些 TODO 勾勒出遗传算法的伪代码，这样就可以在实现过程中标记我们的进展。

我们也借此机会，在 main 方法的顶部初始化一个数组，包含 100 个随机生成的 City 对象。简单地生成随机的 X 和 Y 坐标，将它们传入 City 的构造方法。确保你的 TSP 类像下面这样：

```
package chapter4;
public class TSP {
        public static int maxGenerations = 3000;
        public static void main(String[] args) {

                int numCities = 100;
                City cities[] = new City[numCities];

                // Loop to create random cities
                for (int cityIndex = 0; cityIndex < numCities; cityIndex++) {
                        int xPos = (int) (100 * Math.random());
                        int yPos = (int) (100 * Math.random());
                        cities[cityIndex] = new City(xPos, yPos);
                }

                // Initial GA
                GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001, 0.9, 2, 5);
                // Initialize population
                Population population = ga.initPopulation(cities.length);

                // TODO: Evaluate population


                // Keep track of current generation
                int generation = 1;

                // Start evolution loop
                while (ga.isTerminationConditionMet(generation,
                maxGenerations) == false) {
```

```
            // TODO: Print fittest individual from population

            // TODO: Apply crossover

            // TODO: Apply mutation

            // TODO: Evaluate population


            // Increment the current generation
            generation++;
        }

        // TODO: Display results
    }
}
```

但愿这个过程变得熟悉，我们再次开始实现第 2 章开始时提出的伪代码。我们也生成了 City 对象的数组，准备在评估方法中使用，就像在第 3 章中一样，我们生成了 Maze 对象来评估个体。

剩下的就是生搬硬套：初始化 GeneticAlgorithm 对象（包括种群规模、变异率、交叉率、精英主义计数和锦标赛规模），然后初始化种群。个体的染色体长度，必须与希望访问的城市数量相同。

我们要复用第 3 章中简单的“最大世代数”终止条件，所以现在我们只剩下 6 个 TODO 和工作循环。让我们像往常一样，从评估和适应度评分方法开始。

4.3.4 评估

现在，我们需要评估种群，为个体分配适应度值，这样就知道哪些个体表

现最佳。第一步是为问题定义适应度函数。这里，我们只要计算个体染色体给出的路线的总距离。

首先，我们要创建一个新类，使它能够保存一个路线，并计算出总距离。在包 chapter4 中，创建一个名为 Route 的新类，然后插入以下代码：

```
package chapter4;

public class Route {
	private City route[];
	private double distance = 0;

	public Route(Individual individual, City cities[]) {
		// Get individual's chromosome
		int chromosome[] = individual.getChromosome();
		// Create route
		this.route = new City[cities.length];
		for (int geneIndex = 0; geneIndex < chromosome.length;
		geneIndex++) {
			this.route[geneIndex] = cities[chromosome[geneIndex]];
		}
	}

	public double getDistance() {
		if (this.distance > 0) {
			return this.distance;
		}
```

```
        // Loop over cities in route and calculate route distance
        double totalDistance = 0;
        for (int cityIndex = 0; cityIndex + 1 < this.route.length;
        cityIndex++) {
                totalDistance += this.route[cityIndex].
                distanceFrom(this.route[cityIndex + 1]);
        }

        totalDistance += this.route[this.route.length - 1]
        .distanceFrom(this.route[0]);
        this.distance = totalDistance;

        return totalDistance;
    }
}
```

这个类只包含一个构造方法和一个计算总路径距离的方法。构造方法接受 Individual 和 City 定义的列表（TSP 类的 main 方法中创建的同一个 City 数组）。随后，构造方法按照染色体的顺序，建立一个 City 对象的数组。这种数据结构使得很容易在 getDistance 方法中评估路线总距离。

getDistance 方法遍历路线数组（City 对象的有序数组），调用 City 类的 distanceFrom 方法，依次计算每两个城市之间的距离，同时累计求和。

为了实现这个适应度评分方法，我们需要更新 GeneticAlgorithm 类的 calcFitness 方法。calcFitness 方法应该将距离计算委托给 Route 类，为了做到这一点，它需要接受 City 定义数组并把它传递给 Route 类。

将下面的方法添加到 GeneticAlgorithm 类，可以放在文件的任意位置。

```
public double calcFitness(Individual individual, City cities[]){
    // Get fitness
    Route route = new Route(individual, cities);
    double fitness = 1 / route.getDistance();

    // Store fitness
    individual.setFitness(fitness);
    return fitness;
}
```

在这个方法中，适应度由1除以路线总距离来计算：因此更短的距离具有更高的得分。适应度算好后，它被保存起来，以便再次需要时能快速取得。

现在，我们可以更新 GeneticAlgorithm 类的 evalPopulation 方法，接受 cities 参数，并找到种群中每个个体的适合度。

```
public void evalPopulation(Population population, City cities[]){
    double populationFitness = 0;

    // Loop over population evaluating individuals and summing population
    fitness
    for (Individual individual : population.getIndividuals()) {
        populationFitness += this.calcFitness(individual, cities);
    }

    double avgFitness = populationFitness / population.size();
    population.setPopulationFitness(avgFitness);
}
```

像往常一样，该函数循环遍历种群，并计算出每个个体的适应度。和以往的实现不同，我们计算平均种群适应度，而不是种群的适应度总和（由于我们采用锦标赛选择，而不是轮盘赌选择，所以实际并不需要种群的适应度。如果根本不记录这个值，也不会有什么改变）。

此时，我们可以解决 TSP 类的 main 方法中与评价和显示结果有关的 4 个 TODO。更新 TSP 类，像下面这样。4 个解决了的 TODO 部分是两行 Evaluate population（循环之前和循环内部），循环开始处的 Print fittest individual from population 行，以及在循环之后的 Display results 行。

```
package chapter4;
public class TSP {
    public static int maxGenerations = 3000;
    public static void main(String[] args) {
        int numCities = 100;
        City cities[] = new City[numCities];
        // Loop to create random cities
        for (int cityIndex = 0; cityIndex < numCities; cityIndex++) {
            int xPos = (int) (100 * Math.random());
            int yPos = (int) (100 * Math.random());
            cities[cityIndex] = new City(xPos, yPos);
        }

        // Initial GA
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001,
        0.9, 2, 5);

        // Initialize population
```

```
        Population population = ga.initPopulation(cities.length);

        // Evaluate population
        ga.evalPopulation(population, cities);

        // Keep track of current generation
        int generation = 1;

        // Start evolution loop
        while (ga.isTerminationConditionMet(generation,
        maxGenerations) == false) {
            // Print fittest individual from population
            Route route = new Route(population.getFittest(0),
            cities);
            System.out.println("G"+generation+" Best distance: " +
            route.getDistance());

            // TODO: Apply crossover

            // TODO: Apply mutation

            // Evaluate population
            ga.evalPopulation(population, cities);

            // Increment the current generation
            generation++;
        }
```

```
            // Display results
            System.out.println("Stopped after " + maxGenerations + "
            generations.");
            Route route = new Route(population.getFittest(0), cities);
            System.out.println("Best distance: " + route.getDistance());
        }
    }
```

此时，我们可以单击 Run，循环将走走过场，打印同样的内容 3000 次，但没有变化。这当然在预料之中，我们需要实现剩下的 2 个 TODO 部分：交叉和变异。

4.3.5 终止检查

正如我们已经了解，没有办法知道是否发现了旅行商问题的最优解，除非尝试各种可能的解。这意味着在找到最优解时，我们在此实现中使用的终止检查不能终止，因为它根本没有办法知道。

既然在找到最优解时没有办法停止，我们可以直接允许算法运行若干世代，最后终止，因此我们可以复用第 3 章中 GeneticAlgorithm 类的 isTerminationConditionMet 方法。

但请注意，在这种情况下（最优解无法得知），有许多复杂的终止技术，而不止是简单地设置一个世代数上限。

一种常用的技术是测量种群的适应度随时间的改善。如果种群还在迅速改善，你可能希望让算法继续。一旦种群停止改善，就可以结束演进并展示最好的解。

在类似旅行商问题这样复杂的解空间中，你可能永远找不到全局最优解，但也有许多强大的局部最优解，过程中遇到的平台通常表明你已经发现了这些局部最优解之一。

有几种方法可以测量遗传算法随时间的改善。最简单的方法是测量最佳个体没有改善的连续世代数。如果没有改善的连续世代数已经超过某个阈值，例如 500 代没有改善，则可以停止算法。

对巨大的解空间采用这个简单方法有一个缺点，即你可能会看到不断提高的种群健康，但它可能会非常慢！组合的可能性非常多，导致总能在几十个世代中获得一点改善，所以永远不会真正遇到连续 500 代没有改善的情况。你当然可以不顾改善，设置世代的最大上限。你也可以实施更复杂的技术，如取得不同时间窗口的移动平均值，并将它们进行比较。如果对于几个时间窗口，适应度改善一直呈下降趋势，就停止算法。

但在我们的例子中，我们坚持第 3 章的朴素方法，在本章的结尾，将实现更好的终止条件作为练习留给读者。

4.3.6 交叉

对于旅行商问题，基因和基因在染色体中顺序都非常重要。事实上，对于旅行商问题，特定基因永远不应该在染色体中有多个副本。这会产生无效解，因为对于给定的路线，城市不应被访问一次以上。设想有 3 个城市的情况：A 市、B 市和 C 市。A、B、C 的路线是有效的，但 C、B、C 的路线是无效的：该路线访问 C 市两次，并且从未访问 A 市。因此，要找到并采用一种交叉方法，能产生问题的有效解，这很重要。

在交叉过程中，我们还要考虑亲代染色体的顺序。这是因为，染色体的顺序会影响解的适应度。事实上，只有顺序是重要的。为了更好地理解为什么是这样，请考虑以下两条包含相同的基因的路线为何完全不同。

```
Route 1: A,B,C,D,E
Route 2: C,A,D,B,E
```

我们以前讨论过均匀交叉，然而，均匀交叉方法适用于个体基因的水平，不考虑染色体的顺序。单点和两点交叉方法做得更好，因为它们处理染色体片段，保持这些片段内的顺序。然而单点和两点交叉的问题在于，它们不关心哪些基因加入染色体或从中删除。这意味着我们可能得到无效的解，染色体中包含相同城市的多个引用，或有缺失的城市。

同时解决了这两个问题的交叉方法，是排序交叉。在这种交叉方法中，第一个亲代染色体的一个子集被选中。然后该子集被添加到后代染色体的相同位置，如图 4-2 所示。

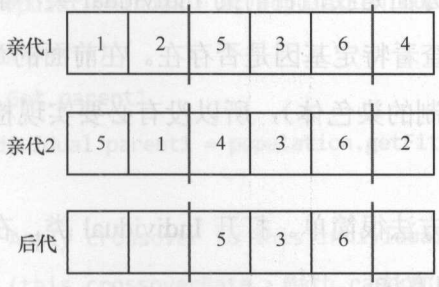


图 4-2　排序交叉

下一步是将第二个亲代的遗传信息添加到后代的染色体中。我们的方法是从所选子集的结束位置开始，然后包括亲代 2 的每个基因，只要后代染色体中还没有该基因。

在这个例子中，我们将从基因 2 开始，检查它是否能在后代的染色体中找到。因为 2 目前不在后代的染色体中，所以可以将它添加到后代染色体的第一个可用位置。然后，因为到达了亲代 2 染色体的末端，所以我们回到第一个基因，5。这一次，5 在后代的染色体中，所以我们跳过它并移动到 1。我们继续这样做，直到得到图 4-3 所示的结果。

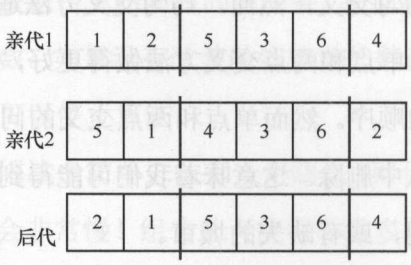


图 4-3 排序交叉示例

这种交叉方法从亲代保留了很多顺序，对于像旅行商这样的问题，也确保解仍然有效。

这个算法有一个方面是我们目前的 Individual 类不能做的：这种技术需要检查后代的染色体，查看特定基因是否存在。在前面的章节中，我们没有特定的基因（我们有二进制的染色体），所以没有必要实现检查一个基因在染色体中存在的方法。

好在，添加这个方法很简单。打开 Individual 类，在任意位置添加一个名为“containsGene”的方法：

```
public boolean containsGene(int gene) {
    for (int i = 0; i < this.chromosome.length; i++) {
        if (this.chromosome[i] == gene) {
            return true;
        }
    }
    return false;
}
```

该方法检查染色体中的每个基因，如果发现它在寻找的基因，就返回 true，否则返回 false。使用该方法解决了这个问题：“这个解访问了城市＃5 吗？让我们调用 individual.containsGene(5)来看看”。

现在，我们准备更新 GeneticAlgorithm 类，在遗传算法中应用排序交叉的方法。像前面的章节一样，我们可以实现锦标赛选择，作为交叉使用的选择方法，但我们还没有修改上一章的 selectParent 方法。

将这个 crossoverPopulation 方法添加到 GeneticAlgorithm 类中：

```
public Population crossoverPopulation(Population population){
        // Create new population
        Population newPopulation = new Population(population.size());

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size();
        populationIndex++) {
                // Get parent1
                Individual parent1 = population.getFittest(populationIndex);

                // Apply crossover to this individual?
                if (this.crossoverRate > Math.random() && populationIndex >=
                this.elitismCount) {
                        // Find parent2 with tournament selection
                        Individual parent2 = this.selectParent(population);

         // Create blank offspring chromosome
                 int offspringChromosome[] = new
                 int[parent1.getChromosomeLength()];
                 Arrays.fill(offspringChromosome, -1);
                Individual offspring = new Individual(offspringChromosome);

                 // Get subset of parent chromosomes
                 int substrPos1 = (int) (Math.random() *
```

```
parent1.getChromosomeLength());
int substrPos2 = (int) (Math.random() *
parent1.getChromosomeLength());

// make the smaller the start and the larger the end
final int startSubstr = Math.min(substrPos1, substrPos2);
final int endSubstr = Math.max(substrPos1, substrPos2);

// Loop and add the sub tour from parent1 to our child
for (int i = startSubstr; i < endSubstr; i++) {
    offspring.setGene(i, parent1.getGene(i));
}

// Loop through parent2's city tour
for (int i = 0; i < parent2.getChromosomeLength(); i++) {
    int parent2Gene = i + endSubstr;
    if (parent2Gene >= parent2.getChromosomeLength()) {
        parent2Gene -= parent2.getChromosomeLength();
    }
    // If offspring doesn't have the city add it
    if (offspring.containsGene
     (parent2.getGene(parent2Gene)) == false) {
        // Loop to find a spare position in the
        child's tour
        for (int ii = 0; ii <
        offspring.getChromosomeLength(); ii++) {
            // Spare position found, add city
            if (offspring.getGene(ii) == -1) {
```

```
                        offspring.setGene(ii,
                        parent2.getGene(parent2Gene));
                        break;
                    }
                }
            }
        }
                    // Add child
                    newPopulation.setIndividual(populationIndex, offspring);
                } else {
                    // Add individual to new population without applying
                    crossover
                    newPopulation.setIndividual(populationIndex, parent1);
                }
        }

        return newPopulation;
    }
```

在这个方法中，我们首先创建一个新的种群来保存后代。然后，遍历当前的种群，顺序是适应度高的个体优先。如果启用精英主义，最初的几个精英个体会跳过，直接添加到新的种群，不作改变。然后根据交叉率，对剩下的个体考虑交叉。如果交叉要应用于个体，用 selectParent 方法选择一个亲代（在这个例子中，selectParent 实现了锦标赛选择，像第 3 章一样），并创建一个新的空白个体。

接着，在亲代 1 的染色体中选择两个随机位置，这两个位置之间的遗传信息子集被添加到后代的染色体中。最后，所需的其余遗传信息按亲代 2 中的顺序添加，完成之后，该个体被添加到新的种群中。

现在，我们可以在 TSP 类的 main 方法中，实现 crossoverPopulation 方法，解决一个 TODO 部分。找到 TODO: Apply crossover 部分，并将其替换为：

```
// Apply crossover
population = ga.crossoverPopulation(population);
```

此时点击 Run，应该得到能工作的算法！3000 代以后，你应该会看到最佳距离大约是 1500。然而，你可能还记得，仅使用交叉容易陷入局部最优，这时你可能会发现算法的停滞时期。我们采用变异的方法，随机地将候选个体扔到解空间的新位置，这有助于改善长期的结果，代价是损失短期结果。

4.3.7 变异

像交叉一样，对旅行商问题采用的交叉类型很重要，因为也要确保交叉应用之后，染色体仍是有效的。随机改变一个基因可能会导致基因在染色体中重复，从而导致染色体无效。

一个简单的解决方案就是“交换变异”。该算法简单地交换两个位置的遗传信息。交换变异的工作原理是循环遍历该个体的染色体的每个基因，根据变异率决定是否变异。如果选中基因进行变异，就在染色体中随机选择另一个基因，然后交换它们的位置，如图 4-4 所示。

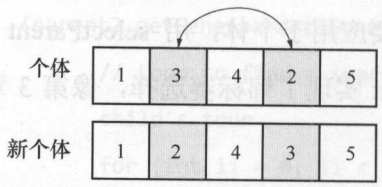


图 4-4 交换变异

这个过程确保没有创建重复的基因，产生的后代都是有效的解。

要实现这种变异方法，先将 mutatePopulation 方法添加到 GeneticAlgorithm 类。

```
public Population mutatePopulation(Population population){
        // Initialize new population
        Population newPopulation = new Population(this.populationSize);

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex <
        population.size(); populationIndex++) {
                Individual individual =
                population.getFittest(populationIndex);

                // Skip mutation if this is an elite individual
                if (populationIndex >= this.elitismCount) {
                        // System.out.println("Mutating population member
                        "+populationIndex);
                        // Loop over individual's genes
                        f or (int geneIndex = 0; geneIndex <
                        individual.getChromosomeLength(); geneIndex++) {
                        // Does this gene need mutation?
                                if (this.mutationRate > Math.random()) {
                                        // Get new gene position
                                        int newGenePos = (int) (Math.random() *
                                        individual.getChromosomeLength());
                                        // Get genes to swap
                                        int gene1 = individual.getGene(newGenePos);
                                        int gene2 = individual.getGene(geneIndex);
                                        // Swap genes
                                        individual.setGene(geneIndex, gene1);
                                        individual.setGene(newGenePos, gene2);
```

```
                }
            }
        }

        // Add individual to population
        newPopulation.setIndividual(populationIndex, individual);
    }

    // Return mutated population
    return newPopulation;
}
```

该方法的第一步是建立一个新种群，来保存变异的个体。然后，从最适应的个体开始遍历种群。如果启用了精英主义，就跳过最初的几个个体，直接添加到新种群中，不作改变。然后，遍历剩余个体的染色体，根据变异率来考虑每个基因是否变异。如果一个基因要变异，就从该个体随机选择另一个基因，彼此交换。最后，将变异的个体加入新种群。

现在，我们可以将变异方法加入 TSP 类的 main 的方法，解决最后一个 TODO 部分。找到注释 TODO: Apply mutation，将其替换为：

```
// Apply mutation
population = ga.mutatePopulation(population);
```

4.3.8 执行

TSP 类的最终代码应该是这样的：

```
package chapter4;
public class TSP {
```

```
public static int maxGenerations = 3000;
public static void main(String[] args) {

    // Create cities
    int numCities = 100;
    City cities[] = new City[numCities];
    // Loop to create random cities
    for (int cityIndex = 0; cityIndex < numCities; cityIndex++) {
        // Generate x,y position
        int xPos = (int) (100 * Math.random());
        int yPos = (int) (100 * Math.random());

        // Add city
        cities[cityIndex] = new City(xPos, yPos);
    }

    // Initial GA
    GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.001,
    0.9, 2, 5);

    // Initialize population
    Population population = ga.initPopulation(cities.length);

    // Evaluate population
    //ga.evalPopulation(population, cities);

    Route startRoute = new Route(population.getFittest(0), cities);
    System.out.println("Start Distance: " + startRoute.getDistance());

    // Keep track of current generation
```

```
        int generation = 1;
        // Start evolution loop
        while (ga.isTerminationConditionMet(generation,
        maxGenerations) == false) {
            // Print fittest individual from population
            Route route = new Route(population.getFittest(0),
            cities);
            System.out.println("G"+generation+" Best distance:
            " + route.getDistance());

            // Apply crossover
            population = ga.crossoverPopulation(population);

            // Apply mutation
            population = ga.mutatePopulation(population);

            // Evaluate population
            ga.evalPopulation(population, cities);

            // Increment the current generation
            generation++;
        }

        System.out.println("Stopped after " + maxGenerations + "
        generations.");
        Route route = new Route(population.getFittest(0), cities);
        System.out.println("Best distance: " + route.getDistance());
    }
}
```

另外，检查 GeneticAlgorithm 类的下列属性和方法签名。如果你遗漏了一个方法的实现，或一个方法的签名不符，那么，现在就回去解决这个问题。

```
package chapter4;
import java.util.Arrays;

public class GeneticAlgorithm {

        private int populationSize;
        private double mutationRate;
        private double crossoverRate;
        private int elitismCount;
        protected int tournamentSize;

        public GeneticAlgorithm(int populationSize, double mutationRate,
        double crossoverRate, int elitismCount, int tournamentSize) { }
        public Population initPopulation(int chromosomeLength){ }
        public boolean isTerminationConditionMet(int generationsCount,
        int maxGenerations) { }
        public double calcFitness(Individual individual, City cities[]) { }
        public void evalPopulation(Population population, City cities[]) { }
        public Individual selectParent(Population population) { }
        public Population crossoverPopulation(Population population) { }
        public Population mutatePopulation(Population population) { }
}
```

最后，确保已经更换了构造方法，添加了 containsGene 方法，更新了 Individual 类。

现在，点击 Run 并观察输出。像往常一样提醒自己，遗传算法是由统计决定的随机过程，你不能仅根据一次运行就做出任何论断。遗憾的是，该问题初

始化了一组随机的城市，意味着该程序的每次运行将有不同的最优解。这让我们很难试验遗传算法的参数并判断它们的表现。本章末的第一个练习是硬编码一组城市，这样就可以准确地评估算法的表现。

但是，此时你会有一些有趣的发现。运行该算法几次，并观察最佳距离。它们是否都相似？至少差距不太大？这很有趣，因为每次运行采用一组不同的城市。但反思之后发现，这是有道理的：虽然城市每次都在不同的位置，但每次都有 100 个城市，并且它们都随机放在 100×100 的地图上，这意味着我们可以很容易地估算问题解的总距离。

考虑 100×100 的地图（它的面积是 10000 单位），目标不是访问 100 个城市，而是 10000 个城市。如果城市均匀放在地图上（每个格点上一个），最佳解的距离应该就是 10100（以 Z 字形访问地图上的每一小块）。如果城市不是均匀分布的，你要随机分布 10000 个城市，最佳解会是以 10000 为中心的统计分布，由于位置的随机性，每次运行结果稍有不同。

现在，我们可以倒推来考虑城市较少的情况。只在地图上均匀地放置 25 个城市，最短路线是 600 单位。这里的关系变得明确：距离与地图面积的平方根乘以城市数量的平方根有关。利用这种关系，我们发现，100 个城市放置均匀时最小距离是 1100（即 $\sqrt{(\text{map})} * \sqrt{(\text{numCities})} + \sqrt{(\text{map})}$，末尾加上的平方根表示南北向的路程，但我们开始从统计上看时，可以舍弃这一项）。如果我们将同样 100 个城市随机放在地图上，可以预期最小距离是以 1000 为中心的分布。类似地，1000 个城市的最佳距离应该接近 3100。

如果变换城市数量，你会发现，对于较小的数字，该算法很容易证实这些推测，但自然，对于超过 100 个城市，它很难找到最小值。

既然我们明白了地图大小、城市数量和预期最佳距离之间的关系，我们甚至可以不用常数个城市来进行试验，并利用统计预期。特别有趣的一个领域是

变异对结果质量的影响。

如果你将解空间想象为许多丘陵连绵起伏的地形，遗传算法就像在地形的随机位置上，扔下 100 个行为不同的人，看看谁发现最低的山谷（在个例子中，因为我们的 Individual 构造方法不是随机的，所以我们实际上将所有人放在了同一地点）。通过一些世代，个体及其后代会向下移动，然后在找到它们附近最低的山谷时停下来。但是，变异将它们拎出来，将它们扔到一个新的随机位置：该位置可能比前一个更好或更坏，但至少这是一个新的、独特的位置，让它们能够继续在新的地形上搜索。

但是，变异常常会有短期的损害。变异可以是有利或不利的，这就是我们采用精英主义的原因：保护最好的个体，避免其参加变异。然而，变异引入的多元化可以产生深远的长期影响，它将个体放到一个地形，否则该地形可能不会被探索：想象一下，一座很高的火山中间有一个巨大的裂口，包含该地形的最低点（全局最优，被不利的地形包围）。任何一个种群都不太可能爬上火山，找到中央的全局最优，除非随机变异将个体放在火山口的边缘。

既然这样说，请观察不同的变异率和精英主义个数对长期运行（代数以万计）的困难问题（200 个城市以上）的影响。在这种情况下，变异有利还是有弊？精英主义有利还是有弊？

4.4 小结

旅行商问题是一个典型的优化问题：对于一组城市，访问每个城市一次，并返回初始的城市，最短的可能路线是什么？

它是一个未解决的优化问题，最优解只能通过暴力算法找到。但是，随着城市的增长，旅行商问题可能解的数量迅速增长，导致暴力算法很快就不可行，即便使用最强大的计算机，也是这样。对于这些情况，人们采用启发式方法，寻找一个很好的近似解，作为代替。

我们介绍了旅行商问题的基本实现，采用二维地图上的城市，用直线距离连接它们。

我们可以用一个有序的城市 ID 列表作为染色体的编码，表示旅行商问题的一个解。但是，由于每个城市 ID 必须在编码中出现至少一次，且只有一次，我们探讨了两种新的交叉和变异方法，可以保持这个约束：排序交叉和交换变异。

在排序交叉中，亲代 1 的染色体的一个随机子集被添加到后代中，然后所需的其余遗传信息按照它在亲代 2 的染色体中的顺序，添加到后代中。这种方法添加了亲代 1 的遗传信息的一个子集，然后只从亲代 2 添加缺少的剩余遗传信息，从而保证了每个城市的 ID 在解中出现且仅出现一次。

在交换变异中，选择两个基因并交换它们的位置。同样，这种变异方法可以保证得到旅行商问题的有效解，因为它既不会丢失一个城市的 ID，也不会导致一个城市出现两次。

4.5 练习

1. 在 TSP 类的 main 方法中硬编码城市，这样你就可以准确地衡量算法的表现。

2．利用用户定义的城市间的距离和时间，添加对最短路线和最快路线的支持。

3．添加非对称 TSP 的支持（从 A 市到 B 市与从 B 市到 A 市的成本可能不同）。

4．修改 Individual 类的构造方法，随机排列城市的顺序。这将如何影响算法的表现？

5．更新终止条件，测量算法的进步，在没有显著进展时退出。这将如何影响算法的表现和结果？

第5章

排课

5.1 简介

本章我们将创建一个遗传算法，为大学排课。我们将研究几种不同的场景，其中可能用到排课算法，并研究设计课表时通常要满足的约束。最后，我们将构建一个简单的排课程序，它可以扩展，支持更复杂的实现。

在人工智能领域，排课是约束满足问题的一个变种。这类问题需要设置一组变量，以便能不违反一组预定义的约束。

约束分为两类：硬约束（必须满足才能得到能工作的解决方案）和软约束（最好满足，但不以牺牲硬约束为代价）。

例如，在制造新产品时，产品的功能性要求是硬约束，并规定了重要的性能要求。不满足这些限制，就没有产品。不能打电话的电话不能算是电话！但你也可能有软约束：虽然不是必需的，但仍是重要的考虑因素，如成本、重量或产品的美感。

创建排课算法时，通常需要考虑许多硬约束和软约束。排课问题中一些典型的硬约束是：

- 教授在任何特定时间只能上一门课；
- 教室要足够大，容纳该班的学生；
- 教室在任何给定的时间只能上一门课；
- 教室必须包含所有需要的设备。

一些典型的软约束可能是：

- 教室的容量应该适合班级的规模；
- 教授喜欢的教室；
- 教授喜欢的上课时间。

有时候，多个软约束可能会发生冲突，需要在它们之间权衡。例如，一个班可能只有 10 名学生，因此软约束可能是希望分配合适的教室，其中有 10 来个座位。但是，上课的教授可能喜欢更大的教室，可以容纳 30 名学生。如果教授的喜好作为软约束，这些配置之一将优先考虑，并希望排课程序能知道。在更高级的实现中，还可以让算法考虑软约束的权重，知道哪个软约束是最重要的，要优先考虑。

像旅行商问题，可以用迭代方法来找到排课问题的最优解。但是，随着班级配置数目的增加，找到最优解就越来越难。在这些情况下，当班级配置的可能数量使迭代方法变得不可行时，遗传算法就是很好的替代。虽然不能保证找到最优解，但它们特别擅长在合理的时间内，寻找接近最优的解。

5.2 问题

我们在本章中探讨的排课问题是一个大学的排课程序，可以根据我们提

供的数据，如可用的教授、可用的教室、时段和学生分组，创建一所大学的时间表。

我们应该注意，创建一所大学的时间表与创建中小学的时间表略有不同。中小学时间表要求学生有全天安排课程的完整时间表，没有空闲时间。相反，典型的大学时间表经常会有自由时间，这取决于学生注册了多少学习单元。

排课程序将为每个班级分配一个时段、一位教授、一个房间和一个学生分组。通过累计学生分组的数目，乘以每个学生分组注册的学习单元数，我们可以计算需要排课的班集总数。

对于应用程序要排课的每个班级，我们会考虑以下硬约束：

- 班级只能安排在空闲的教室里；
- 一位教授在任何时间只能教一个班；
- 教室必须足够大，能容纳学生分组。

为了保持简单，在这个实现中，我们暂时只考虑硬约束。但根据时间表的规则说明书，通常有更多的硬约束。此外，规则说明书也可能包含一些软约束，我们暂时会忽略。虽然不是必要的，但考虑软约束往往会让遗传算法产生的时间表在品质上大为不同。

5.3 实现

现在是时候继续前进，用我们的遗传算法知识来解决问题了。为这个问题建立一个新的 Java/Eclipse 包，我们从编码染色体开始。

5.3.1 开始之前

本章内容基于你在以前所有章节中开发的代码，所以要仔细遵循这一小节的要求，这十分重要！

在开始之前，创建一个新的 Eclipse 或 NetBeans 项目，或在针对本书已有的项目中创建一个新包，名为 chapter5。

从第 4 章复制 Individual、Population 和 GeneticAlgorithm 类，并将它们导入到 chapter5 中。请确保更新每个类文件顶部的包名！在它们的顶部都应该加上 package chapter5。

打开 GeneticAlgorithm 类，并进行以下更改：

- 删除 selectParent 方法，并用 chapter3 的 selectParent 方法（锦标赛选择）取代它；
- 删除 crossoverPopulation 方法，并用 chapter2 的 crossoverPopulation 方法（均匀交叉）取代它；
- 删除 initPopulation、mutatePopulation、evalPopulation 和 calcPopulation 方法，因为你会在本章中重新实现它们。

Population 和 Individual 类可以暂时留下，但请记住，在本章稍后，你将为这两个文件分别添加一个新的构造方法。

5.3.2 编码

我们在排课程序中使用的编码方式，需要能够有效地编码所需的全部班级属性。对于这个实现，它们是：该课安排的时段、该课的教授以及该课的教室。

我们可以为每个时段、教授、教室分配一个数字 ID。然后采用整数数组编码的染色体，这是我们熟悉的方式。这意味着每个需要安排的班只需要 3 个

整数进行编码，如图 5-1 所示。

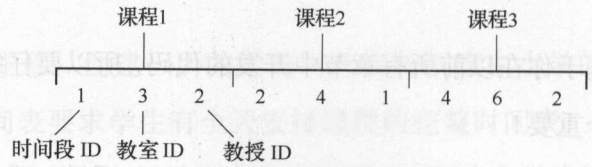


图 5-1 排课示意

将这个数组分成 3 段，就可以取得每个班的所有需要的信息。

5.3.3 初始化

既然我们已经理解了问题，知道了如何编码染色体，就可以开始实现。首先，需要创建一些数据，让排课程序来使用：具体来说，就是房间、教授、时段、学习单元和学生分组，我们试图围绕它们建立一个时间表。

通常这种数据来自一个数据库，包含完整的课程学习单元和学生数据。但是，考虑到本实现的目的，我们将创建一些硬编码的虚拟数据，用于工作。

让我们先建立一些辅助 Java 类。我们会为以上每个数据类型（教室、班级、分组、教授、学习单元和时段）创建一个容器类。这些容器类非常简单，它们大多定义了一些类属性、取值方法和设值方法，没有真正的逻辑。我们在这里依次打印它们。

首先，创建一个 Room 类，保存有关教室的信息。像往常一样，如果使用 Eclipse，你可以用 File ➤ New ➤ Class 菜单项，创建这个类。

```
package chapter5;
public class Room {
        private final int roomId;
        private final String roomNumber;
        private final int capacity;
```

```
    public Room(int roomId, String roomNumber, int capacity) {
        this.roomId = roomId;
        this.roomNumber = roomNumber;
        this.capacity = capacity;
    }

    public int getRoomId() {
        return this.roomId;
    }

    public String getRoomNumber() {
        return this.roomNumber;
    }

    public int getRoomCapacity() {
        return this.capacity;
    }
}
```

该类包含一个构造方法，它接受教室 ID、教室号和教室容量作为参数。它还提供了一些方法，取得教室的属性。

接下来，创建 Timeslot 类，时段表示班级上课在周几的什么时间。

```
package chapter5;
public class Timeslot {
    private final int timeslotId;
    private final String timeslot;

    public Timeslot(int timeslotId, String timeslot){
        this.timeslotId = timeslotId;
```

```
            this.timeslot = timeslot;
        }

        public int getTimeslotId(){
            return this.timeslotId;
        }

        public String getTimeslot(){
            return this.timeslot;
        }
    }
```

时段可以用构造方法来创建，向它传入时段 ID 和时段细节的字符串（细节可能看起来像 Mon 9:00—10:00 这样）。该类也包含一些取值方法，以获取对象的属性。

要建立的第 3 个类是 Professor 类：

```
package chapter5;
public class Professor {
    private final int professorId;
    private final String professorName;

    public Professor(int professorId, String professorName){
        this.professorId = professorId;
        this.professorName = professorName;
    }

    public int getProfessorId(){
        return this.professorId;
```

```
    }

    public String getProfessorName(){
        return this.professorName;
    }
}
```

Professor 类包含一个构造方法，接受教授的 ID 和教授的名字。它也包含一些取值方法，取得教授的属性。

接下来，添加一个 Module 类，保存有关课程学习单元的信息。“学习单元”有时称为“课程”，如“微积分 101”或“美国历史 302”，像现实生活的课程一样，可以有多个部分，参加课程的学生分组在每周的不同时间听不同教授讲课。

```
package chapter5;
public class Module {
    private final int moduleId;
    private final String moduleCode;
    private final String module;
    private final int professorIds[];

    public Module(int moduleId, String moduleCode, String module,
    int professorIds[]){
        this.moduleId = moduleId;
        this.moduleCode = moduleCode;
        this.module = module;
        this.professorIds = professorIds;
    }
```

```
    public int getModuleId(){
        return this.moduleId;
    }

    public String getModuleCode(){
        return this.moduleCode;
    }

    public String getModuleName(){
        return this.module;
    }

    public int getRandomProfessorId(){
        int professorId = professorIds[(int) (professorIds.length *
        Math.random())];
        return professorId;
    }
}
```

这个学习单元类包含一个构造方法，它接受学习单元 ID（数字），学习单元代码（类似“CS101”或“历史 302”），学习单元名称，一组教授的 ID，他们可以教授该学习单元。学习单元类还提供了一些取值方法，以及一个随机选择教授 ID 的方法。

下一个类是 Group，它保存有关学生分组的信息。

```
package chapter5;
public class Group {
```

```
    private final int groupId;
        private final int groupSize;
        private final int moduleIds[];

        public Group(int groupId, int groupSize, int moduleIds[]){
            this.groupId = groupId;
            this.groupSize = groupSize;
            this.moduleIds = moduleIds;
        }

        public int getGroupId(){
            return this.groupId;
        }

        public int getGroupSize(){
            return this.groupSize;
        }

        public int[] getModuleIds(){
            return this.moduleIds;
        }
    }
```

Group 类的构造方法接受分组 ID、分组大小和分组正在上的一组学习单元 ID。它也提供了一些取值方法，获取分组信息。

接下来，添加一个 Class 类。这里的术语可能在整章中带来混乱，因此首字母大写的 Class 指的是你要创建这个 Java 类，我们用小写的 class 指任何其他 Java 类。

Class 类代表所有上述信息的组合。它代表一个学生分组在特定时间，在特定教室，跟随特定教授，学习一个学习单元的一部分。

```
package chapter5;

public class Class {
	private final int classId;
	private final int groupId;
	private final int moduleId;
	private int professorId;
	private int timeslotId;
	private int roomId;

	public Class(int classId, int groupId, int moduleId) {
		this.classId = classId;
		this.moduleId = moduleId;
		this.groupId = groupId;
	}

	public void addProfessor(int professorId) {
		this.professorId = professorId;
	}

	public void addTimeslot(int timeslotId) {
		this.timeslotId = timeslotId;
	}

	public void setRoomId(int roomId) {
		this.roomId = roomId;
```

```
    }

    public int getClassId() {
        return this.classId;
    }

    public int getGroupId() {
        return this.groupId;
    }

    public int getModuleId() {
        return this.moduleId;
    }

    public int getProfessorId() {
        return this.professorId;
    }

    public int getTimeslotId() {
        return this.timeslotId;
    }

    public int getRoomId() {
        return this.roomId;
    }
}
```

现在，我们可以创建一个 Timetable 类，将所有这些对象封装到一个时间表对象中。Timetable 类是至今为止最重要的类，因为它是唯一理解不同的约束彼此之间应该如何相互交互的一个类。

Timetable类也知道如何解析染色体，创建候选的Timetable实例，用于评估和打分。

```
package chapter5;

import java.util.HashMap;

public class Timetable {
	private final HashMap<Integer, Room> rooms;
	private final HashMap<Integer, Professor> professors;
	private final HashMap<Integer, Module> modules;
	private final HashMap<Integer, Group> groups;
	private final HashMap<Integer, Timeslot> timeslots;
	private Class classes[];

	private int numClasses = 0;

	/**
	 * Initialize new Timetable
	 *
	 */
	public Timetable() {
		this.rooms = new HashMap<Integer, Room>();
		this.professors = new HashMap<Integer, Professor>();
		this.modules = new HashMap<Integer, Module>();
		this.groups = new HashMap<Integer, Group>();
		this.timeslots = new HashMap<Integer, Timeslot>();
	}
```

```
public Timetable(Timetable cloneable) {
        this.rooms = cloneable.getRooms();
        this.professors = cloneable.getProfessors();
        this.modules = cloneable.getModules();
        this.groups = cloneable.getGroups();
        this.timeslots = cloneable.getTimeslots();
    }

    private HashMap<Integer, Group> getGroups() {
        return this.groups;
    }

    private HashMap<Integer, Timeslot> getTimeslots() {
        return this.timeslots;
    }

    private HashMap<Integer, Module> getModules() {
        return this.modules;
    }

    private HashMap<Integer, Professor> getProfessors() {
        return this.professors;
    }

    /**
     * Add new room
     *
     * @param roomId
```

```
     * @param roomName
     * @param capacity
     */
    public void addRoom(int roomId, String roomName, int capacity) {
        this.rooms.put(roomId, new Room(roomId, roomName, capacity));
    }

    /**
     * Add new professor
     *
     * @param professorId
     * @param professorName
     */
    public void addProfessor(int professorId, String professorName) {
        this.professors.put(professorId, new Professor(professorId,
        professorName));
    }

    /**
     * Add new module
     *
     * @param moduleId
     * @param moduleCode
     * @param module
     * @param professorIds
     */
    public void addModule(int moduleId, String moduleCode, String
        module, int professorIds[]) {
```

```
        this.modules.put(moduleId, new Module(moduleId, moduleCode,
        module, professorIds));
    }

    /**
     * Add new group
     *
     * @param groupId
     * @param groupSize
     * @param moduleIds
     */
    public void addGroup(int groupId, int groupSize, int moduleIds[]) {
        this.groups.put(groupId, new Group(groupId, groupSize,
        moduleIds));
        this.numClasses = 0;
    }

    /**
     * Add new timeslot
     *
     * @param timeslotId
     * @param timeslot
     */
    public void addTimeslot(int timeslotId, String timeslot) {
        this.timeslots.put(timeslotId, new Timeslot(timeslotId,
        timeslot));
    }
```

```
    /**
     * Create classes using individual's chromosome
     *
     * @param individual
     */
    public void createClasses(Individual individual) {
        // Init classes
        Class classes[] = new Class[this.getNumClasses()];

        // Get individual's chromosome
        int chromosome[] = individual.getChromosome();
        int chromosomePos = 0;
        int classIndex = 0;

        for (Group group : this.getGroupsAsArray()) {
            int moduleIds[] = group.getModuleIds();
            for (int moduleId : moduleIds) {
                classes[classIndex] = new Class(classIndex,
                group.getGroupId(), moduleId);

                // Add timeslot

                classes[classIndex].addTimeslot(chromosome
                [chromosomePos]);
                chromosomePos++;

                // Add room
```

```
                    classes[classIndex].setRoomId(chromosome
                    [chromosomePos]);
                    chromosomePos++;

                    // Add professor

                    classes[classIndex].addProfessor(chromosome
                    [chromosomePos]);
                    chromosomePos++;

                    classIndex++;
                }
            }
            this.classes = classes;
        }

        /**
         * Get room from roomId
         *
         * @param roomId
         * @return room
         */
        public Room getRoom(int roomId) {
            if (!this.rooms.containsKey(roomId)) {
                System.out.println("Rooms doesn't contain key " +
                roomId);
            }
            return (Room) this.rooms.get(roomId);
```

```
    }

    public HashMap<Integer, Room> getRooms() {
        return this.rooms;
    }

    /**
     * Get random room
     *
     * @return room
     */
    public Room getRandomRoom() {
        Object[] roomsArray = this.rooms.values().toArray();
        Room room = (Room) roomsArray[(int) (roomsArray.length *
        Math.random())];
        return room;
    }

    /**
     * Get professor from professorId
     *
     * @param professorId
     * @return professor
     */
    public Professor getProfessor(int professorId) {
        return (Professor) this.professors.get(professorId);
    }
```

```
/**
 * Get module from moduleId
 *
 * @param moduleId
 * @return module
 */
public Module getModule(int moduleId) {
        return (Module) this.modules.get(moduleId);
}

/**
 * Get moduleIds of student group
 *
 * @param groupId
 * @return moduleId array
 */
public int[] getGroupModules(int groupId) {
        Group group = (Group) this.groups.get(groupId);
        return group.getModuleIds();
}

/**
 * Get group from groupId
 *
 * @param groupId
 * @return group
 */
public Group getGroup(int groupId) {
```

```
        return (Group) this.groups.get(groupId);
    }

    /**
     * Get all student groups
     *
     * @return array of groups
     */
    public Group[] getGroupsAsArray() {
        return (Group[]) this.groups.values().toArray(new
        Group[this.groups.size()]);
    }

    /**
     * Get timeslot by timeslotId
     *
     * @param timeslotId
     * @return timeslot
     */
    public Timeslot getTimeslot(int timeslotId) {
        return (Timeslot) this.timeslots.get(timeslotId);
    }

    /**
     * Get random timeslotId
     *
     * @return timeslot
     */
```

```
public Timeslot getRandomTimeslot() {
        Object[] timeslotArray = this.timeslots.values().toArray();
        Timeslot timeslot = (Timeslot) timeslotArray[(int)
         (timeslotArray.length * Math.random())];
        return timeslot;
}

/**
 * Get classes
 *
 * @return classes
 */
public Class[] getClasses() {
        return this.classes;
}

/**
 * Get number of classes that need scheduling
 *
 * @return numClasses
 */
public int getNumClasses() {
        if (this.numClasses > 0) {
                return this.numClasses;
        }

        int numClasses = 0;
        Group groups[] = (Group[]) this.groups.values().toArray(new
        Group[this.groups.size()]);
```

```
        for (Group group : groups) {
            numClasses += group.getModuleIds().length;
        }
        this.numClasses = numClasses;

        return this.numClasses;
    }

    /**
     * Calculate the number of clashes
     *
     * @return numClashes
     */
    public int calcClashes() {
        int clashes = 0;

        for (Class classA : this.classes) {
            // Check room capacity
            int roomCapacity = this.getRoom(classA.getRoomId()).
            getRoomCapacity();
            int groupSize = this.getGroup(classA.getGroupId()).
            getGroupSize();
            if (roomCapacity < groupSize) {
                clashes++;
            }

            // Check if room is taken
            for (Class classB : this.classes) {
                if (classA.getRoomId() == classB.getRoomId()
```

```
                    && classA.getTimeslotId() == classB.
                    getTimeslotId()
                                && classA.getClassId() !=
                                classB.getClassId()) {
                            clashes++;
                            break;
                        }
                    }

                    // Check if professor is available
                    for (Class classB : this.classes) {
                        if (classA.getProfessorId() == classB.
                        getProfessorId() && classA.getTimeslotId() ==
                        classB.getTimeslotId()
                                && classA.getClassId() !=
                                classB.getClassId()) {
                            clashes++;
                            break;
                        }
                    }
                }
                return clashes;
            }
        }
```

这个类包含一些方法，将教室、时段、教授、学习单元和学生分组添加到时间表中。通过这种方式，Timetable 类有双重目的：一个 Timetable 对象知道所有可用的教室、时段、教授等，但 Timetable 对象也可以读取染色体，从染色体创建类的子集，帮助评估染色体的适应度。

请密切关注这个类中的两个重要方法：createClasses 和 calcClashes。

createClasses 方法接受一个 Individual（即一条染色体），并利用它获知学生分组总数和必须排课的学习单元总数，为这些分组和单元创建一些 Class 对象。然后，该方法开始读取染色体，并将可变的信息（时段、教室、教授）分配给这些类的每一个。因此，createClasses 方法确保考虑到每个单元和学生分组，但它利用遗传算法和得到的染色体来尝试时段、教室和教授的不同组合。Timetable 类将这些信处缓存在本地（作为 this.classes），以备稍后使用。

Classes 建好后，calcClashes 方法依次检查每一个，统计“冲突”的数量。在本例中，“冲突”是违反任何硬约束，例如班级的教室太小、教室和时段冲突，或教授和时段冲突。冲突的数量稍后由 GeneticAlgorithm 的 calcFitness 方法使用。

5.3.4 执行类

现在，我们可以创建一个执行类，包含该程序的 main 方法。像前几章一样，我们基于第 2 章的伪代码来构建这个类，其中包含一些 TODO 注释，在本章中将由具体的实现来替代。

首先，创建一个新的 Java 类，名为 TimetableGA。请确保它在 package chapter5 中，并为它添加以下代码：

```
package chapter5;

public class TimetableGA {

    public static void main(String[] args) {
        // TODO: Create Timetable and initialize with all the available
```

```
courses, rooms, timeslots, professors, modules, and groups

// Initialize GA
GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.9, 2, 5);

// TODO: Initialize population

// TODO: Evaluate population

// Keep track of current generation
int generation = 1;

// Start evolution loop
        // TODO: Add termination condition
while (false) {
        // Print fitness
        System.out.println("G" + generation + " Best fitness: " +
        population.getFittest(0).getFitness());

        // Apply crossover
        population = ga.crossoverPopulation(population);

        // TODO: Apply mutation

        // TODO: Evaluate population

        // Increment the current generation
        generation++;
```

```
            }

            // TODO: Print final fitness

            // TODO: Print final timetable
        }
    }
```

为了完成本章，我们留了 8 个 TODO 部分。请注意，交叉不是一个 TODO 部分，我们打算复用第 3 章的锦标赛选择，以及第 2 章的均匀交叉。

第一个 TODO 部分很容易解决，我们现在就会做。一般来说，一所学校的排课信息来自数据库，但我们暂时硬编码一些班级和教授。由于下面的代码有点冗长，我们为它在 TimetableGA 类中创建一个单独的方法。用户可以在任意位置添加这个方法：

```
private static Timetable initializeTimetable() {
    // Create timetable
    Timetable timetable = new Timetable();

    // Set up rooms
    timetable.addRoom(1, "A1", 15);
    timetable.addRoom(2, "B1", 30);
    timetable.addRoom(4, "D1", 20);
    timetable.addRoom(5, "F1", 25);

    // Set up timeslots
    timetable.addTimeslot(1, "Mon 9:00 - 11:00");
    timetable.addTimeslot(2, "Mon 11:00 - 13:00");
```

```
timetable.addTimeslot(3, "Mon 13:00 - 15:00");
timetable.addTimeslot(4, "Tue 9:00 - 11:00");
timetable.addTimeslot(5, "Tue 11:00 - 13:00");
timetable.addTimeslot(6, "Tue 13:00 - 15:00");
timetable.addTimeslot(7, "Wed 9:00 - 11:00");
timetable.addTimeslot(8, "Wed 11:00 - 13:00");
timetable.addTimeslot(9, "Wed 13:00 - 15:00");
timetable.addTimeslot(10, "Thu 9:00 - 11:00");
timetable.addTimeslot(11, "Thu 11:00 - 13:00");
timetable.addTimeslot(12, "Thu 13:00 - 15:00");
timetable.addTimeslot(13, "Fri 9:00 - 11:00");
timetable.addTimeslot(14, "Fri 11:00 - 13:00");
timetable.addTimeslot(15, "Fri 13:00 - 15:00");

// Set up professors
timetable.addProfessor(1, "Dr P Smith");
timetable.addProfessor(2, "Mrs E Mitchell");
timetable.addProfessor(3, "Dr R Williams");
timetable.addProfessor(4, "Mr A Thompson");

// Set up modules and define the professors that teach them
timetable.addModule(1, "cs1", "Computer Science", new int[] { 1, 2 });
timetable.addModule(2, "en1", "English", new int[] { 1, 3 });
timetable.addModule(3, "ma1", "Maths", new int[] { 1, 2 });
timetable.addModule(4, "ph1", "Physics", new int[] { 3, 4 });
timetable.addModule(5, "hi1", "History", new int[] { 4 });
timetable.addModule(6, "dr1", "Drama", new int[] { 1, 4 });
```

```
        // Set up student groups and the modules they take.
        timetable.addGroup(1, 10, new int[] { 1, 3, 4 });
        timetable.addGroup(2, 30, new int[] { 2, 3, 5, 6 });
        timetable.addGroup(3, 18, new int[] { 3, 4, 5 });
        timetable.addGroup(4, 25, new int[] { 1, 4 });
        timetable.addGroup(5, 20, new int[] { 2, 3, 5 });
        timetable.addGroup(6, 22, new int[] { 1, 4, 5 });
        timetable.addGroup(7, 16, new int[] { 1, 3 });
        timetable.addGroup(8, 18, new int[] { 2, 6 });
        timetable.addGroup(9, 24, new int[] { 1, 6 });
        timetable.addGroup(10, 25, new int[] { 3, 4 });
        return timetable;
    }
```

现在，解决在 main 方法顶部的第一个 TODO 部分，用以下代码代替它。

```
// Get a Timetable object with all the available information.
Timetable timetable = initializeTimetable();
```

main 方法的顶部现在看起来应该如下所示。

```
public class TimetableGA {

    public static void main(String[] args) {
        // Get a Timetable object with all the available information.
        Timetable timetable = initializeTimetable();

        // Initialize GA ... (and the rest of the class, unchanged
        from before!)
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.9, 2, 5);
```

我们得到了带有所有必要信息的 Timetable 实例，我们创建的 GeneticAlgorithm 对象类似于前几章：一个遗传算法、种群 100、变异率 0.01、交叉率 0.9、2 个精英个体和锦标赛规模是 5。

我们现在剩下 7 个 TODO 部分。下一个 TODO 部分涉及初始化种群。为了创建一个群，我们需要知道所需染色体的长度，这由 Timetable 中的分组和学习单元数来确定。

我们需要能够从 Timetable 对象初始化一个 Population，这意味着我们也需要能够从 Timetable 对象初始化一个 Individual。因此，为了解决这个 TODO 部分，我们必须做 3 件事：为 GeneticAlgorithm 类添加一个 initPopulation(Timetable)，为 Population 添加一个接受 Timetable 的构造方法，为 Individual 添加一个接受 Timetable 的构造方法。

让我们自底向上开始工作。更新 Individual 类，添加一个新的构造方法，通过 Timetable 构建一个 Individual。该构造方法利用 Timetable 对象来确定必须排课的班级数，这决定了染色体的长度。染色体本身是从 Timetable 随机取得教室、时段和教授而生成的。

将下面的方法添加到 Individual 类的任意位置：

```
public Individual(Timetable timetable) {
        int numClasses = timetable.getNumClasses();

        // 1 gene for room, 1 for time, 1 for professor
        int chromosomeLength = numClasses * 3;
        // Create random individual
        int newChromosome[] = new int[chromosomeLength];
        int chromosomeIndex = 0;
```

```
        // Loop through groups
        for (Group group : timetable.getGroupsAsArray()) {
            // Loop through modules
            for (int moduleId : group.getModuleIds()) {
                // Add random time
                int timeslotId = timetable.getRandomTimeslot().
                getTimeslotId();
                newChromosome[chromosomeIndex] = timeslotId;
                chromosomeIndex++;

                // Add random room
                int roomId = timetable.getRandomRoom().getRoomId();
                newChromosome[chromosomeIndex] = roomId;
                chromosomeIndex++;

                // Add random professor
                Module module = timetable.getModule(moduleId);
                newChromosome[chromosomeIndex] = module.
                getRandomProfessorId();
                chromosomeIndex++;
            }
        }
        this.chromosome = newChromosome;
    }
```

这个构造方法接受一个时间表对象，并遍历每个学生分组和该组注册的每个学习单元（给出需要排课的 Class 总数）。对于每个 Class，随机的教室、教授和时段被选中，相应的 ID 被添加到染色体中。

接下来，将下面的构造方法添加到 Population 类。该构造方法调用我们刚创建的 Individual 构造方法，通过 Timetable 初始化一些 Individual，建立一个 Population。

```
public Population(int populationSize, Timetable timetable) {
        // Initial population
        this.population = new Individual[populationSize];

        // Loop over population size
        for (int individualCount = 0; individualCount < populationSize;
        individualCount++) {
                // Create individual
                Individual individual = new Individual(timetable);
                // Add individual to population
                this.population[individualCount] = individual;
        }
}
```

接下来，重新实现 GeneticAlgorithm 类的 initPopulation 方法，使用新的 Population 构造方法：

```
public Population initPopulation(Timetable timetable) {
        // Initialize population
        Population population = new Population(this.populationSize, timetable);
        return population;
}
```

我们终于可以解决下一个 TODO 部分：替代执行类 main 方法中的 TODO: Initialize Population，调用 GeneticAlgorithm 的 initPopulation 方法：

```
        // Initialize population
        Population population = ga.initPopulation(timetable);
```

执行类 TimetableGA 的 main 方法现在看起来应该像下面这样。因为我们还没有实现终止条件，该代码不会做什么有趣的事。实际上，Java 编译器可能会抱怨循环内无法执行到的代码。我们会很快解决这个问题。

```
public static void main(String[] args) {
        // Get a Timetable object with all the available information.
        Timetable timetable = initializeTimetable();

        // Initialize GA
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.9, 2, 5);

        // Initialize population
        Population population = ga.initPopulation(timetable);

        // TODO: Evaluate population

        // Keep track of current generation
        int generation = 1;

        // Start evolution loop
        // TODO: Add termination condition
        while (false) {
              // Print fitness
              System.out.println("G" + generation + " Best fitness: " +
              population.getFittest(0).getFitness());
```

```
            // Apply crossover
            population = ga.crossoverPopulation(population);

            // TODO: Apply mutation

            // TODO: Evaluate population

            // Increment the current generation
            generation++;
        }

        // TODO: Print final fitness

        // TODO: Print final timetable
    }
```

5.3.5 评估

初始种群已经建立，我们需要评估这些个体并赋予其适应度值。我们从前面知道，优化班级时间表的目标，是尽可能不要打破约束条件。这意味着，个体的适应度值与违反多少约束条件成反比。

打开并检查 Timetable 类的 createClasses 的方法。它知道所有需要排入教室的 Groups 和 Modules，以及教授和特定的时段。利用这些知识，它将染色体转换成 Class 对象的数组，并将它们存放起来用于评估。此方法不进行任何实际的评估，但它是染色体和评估步骤之间的桥梁。

接下来，检查同一个类中的 calcClashes 的方法。这个方法比较每个班级和其他所有班级，如果违反了硬约束，就增加一个冲突。例如，如果选择的教室太小，如果该教室的安排有冲突，或者如果教授的安排有冲突。该方法返回它发现冲突总数。

现在一切已经就绪，可以创建适应度函数，并最终评估种群中个体的适应度。

打开 GeneticAlgorithm 类，先添加下面的 calcFitness 方法。

```
public double calcFitness(Individual individual, Timetable timetable) {

        // Create new timetable object to use -- cloned from an existing
        timetable
        Timetable threadTimetable = new Timetable(timetable);
        threadTimetable.createClasses(individual);

        // Calculate fitness
        int clashes = threadTimetable.calcClashes();
        double fitness = 1 / (double) (clashes + 1);
        individual.setFitness(fitness);

        return fitness;
}
```

calcFitness 方法克隆了传给它的 Timetable 对象，调用了 createClasses 方法，然后通过 calcClashes 方法计算冲突的数量。适应度定义为冲突数量加 1 的倒数：0 个冲突将导致适应度为 1。

再为 GeneticAlgorithm 类添加 evalPopulation 方法。像前面几章一样，该方法只是迭代遍历种群，并对每个个体调用 calcFitness。

```
public void evalPopulation(Population population, Timetable timetable) {
    double populationFitness = 0;

    // Loop over population evaluating individuals and summing
    population
    // fitness
    for (Individual individual : population.getIndividuals()) {
        populationFitness += this.calcFitness(individual, timetable);
    }

    population.setPopulationFitness(populationFitness);
}
```

最后，我们可以评估种群并解决执行类 TimetableGA 的 main 方法中的一些 TODO 部分。将两个 TODO: Evaluate Population 部分更新为：

```
// Evaluate population
ga.evalPopulation(population, timetable);
```

此时，应该剩下 4 个 TODO 部分。程序现在还不能运行，因为终止条件没有定义，循环尚未启用。

5.3.6 终止

编写排课程序的下一步，就是建立终止检查。以前，我们用过世代数和适应度来决定是否终止遗传算法。这一次，我们将结合这两种终止条件，或者若干世代之后，或者找到一个有效解，就终止遗传算法。

由于适应度值是基于打破约束的数目，我们知道，完美解的适应度值是 1。保留以前的终止检查不变，在 GeneticAlgorithm 类中加入这个条件，作为第二

种终止检查。在执行循环中，我们将同时使用这两种检查。

```
public boolean isTerminationConditionMet(Population population) {
        return population.getFittest(0).getFitness() == 1.0;
}
```

此时，确认第二个isTerminationConditionMet方法（应该已经在GeneticAlgorithm类中）像下面这样：

```
public boolean isTerminationConditionMet(int generationsCount, int
maxGenerations) {
            return (generationsCount > maxGenerations);
}
```

现在，我们可以在 main 方法中加入两个终止检查，支持进化循环。打开执行类 TimetableGA，像下面这样解决 TODO: Add termination condition 部分：

```
// Start evolution loop
while (ga.isTerminationConditionMet(generation, 1000) == false
       && ga.isTerminationConditionMet(population) == false) {

          // Rest of the loop in here...
```

第一个 isTerminationConditionMet 调用限制进化 1000 代，第二个检查种群中是否有适应度为 1 的个体。

让我们再快速解决两个 TODO。在循环结束时，有一些简单的报告要显示。删除循环之后的两个 TODO 部分（Print final fitness 和 Print final timetable），用以下代码替换它们：

```
// Print fitness
```

```
timetable.createClasses(population.getFittest(0));
System.out.println();
System.out.println("Solution found in " + generation + " generations");
System.out.println("Final solution fitness: " + population.getFittest(0).
getFitness());
System.out.println("Clashes: " + timetable.calcClashes());

// Print classes
System.out.println();
Class classes[] = timetable.getClasses();
int classIndex = 1;
for (Class bestClass : classes) {
        System.out.println("Class " + classIndex + ":");
        System.out.println("Module: " +
timetable.getModule(bestClass.getModuleId()).getModuleName());
        System.out.println("Group: " +
timetable.getGroup(bestClass.getGroupId()).getGroupId());
        System.out.println("Room: " +
timetable.getRoom(bestClass.getRoomId()).getRoomNumber());
        System.out.println("Professor: " +
timetable.getProfessor(bestClass.getProfessorId()).getProfessorName());
        System.out.println("Time: " +
timetable.getTimeslot(bestClass.getTimeslotId()).getTimeslot());
        System.out.println("-----");
        classIndex++;
}
```

此时，你应该能够运行这个程序，查看进化循环，并且得到一个结果。如果没有变异，可能永远也找不到一个解，但现在重新利用第 2 章、第 3 章中的

交叉方法，往往足以找到一个解。然而，如果运行几次程序，在 1000 代内都不能找到解，你可能要重新阅读本章，以确保没有犯任何错误。

本章我们省略熟悉的“交叉”小节，因为没有新的技术要展示。回想一下，第 2 章的均匀交叉随机选择染色体，与一个亲代交换，不保留基因组内的任何连续性。对于这个问题，这是个好方法。因为在这个例子中，基因分组（代表教授、教室和时段的组合）更可能有害，而不是有利。

5.3.7 变异

回想一下，染色体上的限制，往往决定了遗传算法选择的变异和交叉技术。在本例中，染色体由特定的教室、教授和时段编号构成，我们不能简单地选择随机数。此外，由于教室、教授和时段都有不同范围的 ID，我们也不能简单地选择 1 和“X”之间的随机数。似乎，我们可以针对每个不同类型对象的编码（教室、教授和时段）选择随机数，但这也需要假定 ID 是连续的，而实际上它们可能不是！

我们可以从均匀交叉获得一点启示，来解决变异的问题。在均匀交叉中，基因从现有的、有效的亲代中随机选择。亲代可能不是种群中最好的个体，但至少它是有效的。

变异可以用类似的方式来实现。我们不是选择一个随机数作为染色体中的随机基因，而是创建一个新的、随机的、但有效的个体，本质上是通过均匀交叉来实现变异！也就是说，我们可以用 Individual(Timetable)构造方法创建一个全新的随机 Individual，然后从这个随机 Individual 选择一些基因，直到要变异的 Individual。这种技术称为“均匀变异”，它确保了所有的变异个体完全有效，不会选择没有意义的基因。用户可以在 GeneticAlgorithm 类的任意位置添加下面的方法：

```
public Population mutatePopulation(Population population, Timetable
timetable) {
        // Initialize new population
        Population newPopulation = new Population(this.populationSize);

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size();
        populationIndex++) {
                Individual individual = population.
                getFittest(populationIndex);

                // Create random individual to swap genes with
                Individual randomIndividual = new Individual(timetable);

                // Loop over individual's genes
                for (int geneIndex = 0; geneIndex < individual.
                getChromosomeLength(); geneIndex++) {
                        // Skip mutation if this is an elite individual
                      if (populationIndex > this.elitismCount) {
                              // Does this gene need mutation?
                              if (this.mutationRate > Math.random()) {
                                      // Swap for new gene
                                      individual.setGene(geneIndex,
                                      randomIndividual.getGene(geneIndex));
                              }
                      }
                }
```

```
                // Add individual to population
                newPopulation.setIndividual(populationIndex, individual);
            }

            // Return mutated population
            return newPopulation;
        }
```

在这种方法中，像前几章的变异一样，通过循环遍历种群的非精英个体来实现种群变异。不像其他变异技术通常会直接修改基因，这种变异算法产生一个随机但有效的个体，并从中随机拷贝基因。

现在，我们可以解决执行类的 main 方法中最后一个 TODO 部分。在主循环中加上这一行：

```
// Apply mutation
population = ga.mutatePopulation(population, timetable);
```

现在应该一切就绪，可以运行遗传算法，创建一个新的大学时间表。如果你的 Java IDE 显示错误，或者现在它无法编译，请回顾本章，解决你发现的任何问题。

5.3.8 执行

确保你的 TimetableGA 类如下所示：

```
package chapter5;

public class TimetableGA {

    public static void main(String[] args) {
```

```
        // Get a Timetable object with all the available information.
        Timetable timetable = initializeTimetable();

        // Initialize GA
        GeneticAlgorithm ga = new GeneticAlgorithm(100, 0.01, 0.9, 2, 5);

        // Initialize population
        Population population = ga.initPopulation(timetable);

        // Evaluate population
        ga.evalPopulation(population, timetable);

        // Keep track of current generation
        int generation = 1;

        // Start evolution loop
        while (ga.isTerminationConditionMet(generation, 1000) == false
            && ga.isTerminationConditionMet(population) == false) {
            // Print fitness
            System.out.println("G" + generation + " Best fitness: " +
            population.getFittest(0).getFitness());

            // Apply crossover
            population = ga.crossoverPopulation(population);

            // Apply mutation
            population = ga.mutatePopulation(population, timetable);

            // Evaluate population
            ga.evalPopulation(population, timetable);
```

```
            // Increment the current generation
            generation++;
        }

        // Print fitness
        timetable.createClasses(population.getFittest(0));
        System.out.println();
        System.out.println("Solution found in " + generation + "
        generations");
        System.out.println("Final solution fitness: " + population.
        getFittest(0).getFitness());
        System.out.println("Clashes: " + timetable.calcClashes());

        // Print classes
        System.out.println();
        Class classes[] = timetable.getClasses();
        int classIndex = 1;
        for (Class bestClass : classes) {
            System.out.println("Class " + classIndex + ":");
            System.out.println("Module: " +
                    timetable.getModule(bestClass.getModuleId()).
                    getModuleName());
            System.out.println("Group: " +
                    timetable.getGroup(bestClass.getGroupId()).
                    getGroupId());
            System.out.println("Room: " +
                    timetable.getRoom(bestClass.getRoomId()).
                    getRoomNumber());
            System.out.println("Professor: " +
```

```
                timetable.getProfessor(bestClass.getProfessorId()).
                    getProfessorName());
            System.out.println("Time: " +
                timetable.getTimeslot(bestClass.getTimeslotId()).
                    getTimeslot());
            System.out.println("-----");
            classIndex++;
        }
    }

    /**
     * Creates a Timetable with all the necessary course information.
     * @return
     */
    private static Timetable initializeTimetable() {
        // Create timetable
        Timetable timetable = new Timetable();

        // Set up rooms
        timetable.addRoom(1, "A1", 15);
        timetable.addRoom(2, "B1", 30);
        timetable.addRoom(4, "D1", 20);
        timetable.addRoom(5, "F1", 25);

        // Set up timeslots
        timetable.addTimeslot(1, "Mon 9:00 - 11:00");
        timetable.addTimeslot(2, "Mon 11:00 - 13:00");
        timetable.addTimeslot(3, "Mon 13:00 - 15:00");
        timetable.addTimeslot(4, "Tue 9:00 - 11:00");
        timetable.addTimeslot(5, "Tue 11:00 - 13:00");
```

```
timetable.addTimeslot(6, "Tue 13:00 - 15:00");
timetable.addTimeslot(7, "Wed 9:00 - 11:00");
timetable.addTimeslot(8, "Wed 11:00 - 13:00");
timetable.addTimeslot(9, "Wed 13:00 - 15:00");
timetable.addTimeslot(10, "Thu 9:00 - 11:00");
timetable.addTimeslot(11, "Thu 11:00 - 13:00");
timetable.addTimeslot(12, "Thu 13:00 - 15:00");
timetable.addTimeslot(13, "Fri 9:00 - 11:00");
timetable.addTimeslot(14, "Fri 11:00 - 13:00");
timetable.addTimeslot(15, "Fri 13:00 - 15:00");

// Set up professors
timetable.addProfessor(1, "Dr P Smith");
timetable.addProfessor(2, "Mrs E Mitchell");
timetable.addProfessor(3, "Dr R Williams");
timetable.addProfessor(4, "Mr A Thompson");

// Set up modules and define the professors that teach them
timetable.addModule(1, "cs1", "Computer Science",
new int[] { 1, 2 });
timetable.addModule(2, "en1", "English", new int[] { 1, 3 });
timetable.addModule(3, "ma1", "Maths", new int[] { 1, 2 });
timetable.addModule(4, "ph1", "Physics", new int[] { 3, 4 });
timetable.addModule(5, "hi1", "History", new int[] { 4 });
timetable.addModule(6, "dr1", "Drama", new int[] { 1, 4 });

// Set up student groups and the modules they take.
timetable.addGroup(1, 10, new int[] { 1, 3, 4 });
timetable.addGroup(2, 30, new int[] { 2, 3, 5, 6 });
timetable.addGroup(3, 18, new int[] { 3, 4, 5 });
```

```
            timetable.addGroup(4, 25, new int[] { 1, 4 });
            timetable.addGroup(5, 20, new int[] { 2, 3, 5 });
            timetable.addGroup(6, 22, new int[] { 1, 4, 5 });
            timetable.addGroup(7, 16, new int[] { 1, 3 });
            timetable.addGroup(8, 18, new int[] { 2, 6 });
            timetable.addGroup(9, 24, new int[] { 1, 6 });
            timetable.addGroup(10, 25, new int[] { 3, 4 });
            return timetable;
        }
    }
```

就这样运行排课程序，应该在大约 50 代中产生一个解，并且在所有情况下都应得到零冲突（硬约束）的解。如果算法多次达到 1000 代的上限，或者它提供的解包含冲突，则你的实现可能有错！

花一分钟目测一下算法返回的时间表结果。确认教授、教室和时段之间没有实际的冲突。

此时，你可能想在 TimetableGA 的 initializeTimetable 方法中初始化时间表时，增加更多的教授、学习单元、时段，学生分组和教室。你能强制算法失败吗？

5.4 分析和改进

排课问题是一个很好的例子，它利用遗传算法搜索解空间，寻找有效解，而不是最优解。这个问题可以有很多适应度为 1 的解，而我们要做的只是找到其中一个有效解。如果只考虑硬约束，任意两个有效解之间没有真正的区别，我们可以简单地选择找到的第一个解。

不像第4章的旅行商问题，排课问题的这一特性意味着算法实际上可能返回无效的解。如果旅行商问题访问每个城市不止一次，那么它的解可能无效。但是因为我们非常小心地设计了初始化、交叉和变异算法，所以永远不会从第4章的代码中产生无效解。TSP求解程序返回的路线都是有效的，它只是找到最短路线的问题。如果我们在任何时候停止TSP算法，在任何世代停止，随机从种群中挑选一个成员，它都是有效的解。

但在本章中，大多数解是无效的，只有找到第一个有效解或时间用完时，我们才会停止。这两个问题之间的区别如下：在旅行商问题中，可以很容易地创建一个有效解（只确保每个城市都访问一次，但不保证该解的适应度），但在排课程序中，创建一个有效解是比较困难的部分。

而且，如果没有任何软约束，排课程序返回的任意两个有效解的适应度都没有区别。在这种情况下，硬约束决定解是否有效，但软约束决定解的品质。上述实现不偏好任何特定的有效解，因为它没有办法确定解的品质，只知道解是否有效。

为排课程序添加软约束，将显著改变该问题。我们不再只寻找任意的有效解，而是想要最好的有效解。

好在遗传算法特别擅长这种类型的约束戏法。个体只由一个数字（它的适应度）来评判，这一事实成了我们的优势。确定个体适应度的算法对于基因算法是完全不透明的：对于遗传算法而言，它是一个黑盒子。虽然适应度值对于遗传算法非常重要并且不能随意实现，但它简单而不透明，让我们能够调和各种限制和条件。因为一切都归结为一个单一的、无量纲的适应度值，所以我们就能够随意扩展并变换许多约束，而约束的重要性由它对适应度值的贡献大小表示。

上面实现的排课程序只使用硬约束，将适应度值限制在0～1的范围内。如果组合不同类型的约束，应确保硬约束对适应度值有压倒性的影响，而软约束做出不太大的贡献。

举一个例子，假设你需要为排课程序添加一些软约束，每个的重要性稍微不同。当然，硬约束仍然适用。你如何调和硬约束和软约束？现有的适应度值1 / (clashes+1)显然不包括软约束，即使认为破坏软约束是一次“冲突”，仍然是将它们等同于硬约束。根据该模型，有可能选择一个无效解，因为它可能满足一些软约束，弥补破坏硬约束损失的适应度。

作为替代，请考虑一个新的适应度计分系统：每破坏一个硬约束就从适应度计分中减去100分，而满足任何一个软约束可以增加1分、2分或3分，这取决于它的重要性。根据这个方案，我们只考虑零分以上的解，因为负分表示破坏了硬约束。该方案还确保没有办法通过满足大量的软约束来抵消破坏的硬约束：硬约束适应度得分的贡献非常强大，软约束没有办法弥补破坏硬约束的100分扣除。最后，该方案也可以让你排列软约束的优先级，更重要的约束可以贡献更多的适应度得分。

为了进一步说明适应度计分让约束标准化的思想，请考虑任一个地图和导航工具（如谷歌地图）。当你搜索两个地点之间的导航，对适应度得分的主要贡献是起点到终点所花的时间。一个简单的算法可能使用行程的分钟数作为适应度得分（在本例中，我们称之为成本得分，因为越低越好，适应度反过来通常称为成本）。

需要60分钟比需要70分钟的路线更好，但我们知道这并非总是现实生活中的情况。也许较短的路线有20元钱的高昂通行费。用户可以选择“避免通行费”选项，现在算法必须调和驾驶时间和通行费用。多少分钟值一美元？如果你决定每一美元增加了一个点的成本得分，较短的路线现在成本是80，输给了较长但更便宜的路线。另一方面，如果避免通行费的权重较少，决定一美元的通行费只为路线增加0.25的成本，较短的路线仍然会以65的成本取胜。

最后，在遗传算法中处理硬约束和软约束时，一定要明白适应度得分代表什么，以及每个约束如何影响个体的得分。

5.5 小结

在本章中，我们介绍了利用遗传算法来排课的基本知识。不是利用遗传算法找到一个最佳解，而是利用遗传算法找到满足若干硬约束的第一个有效解。

我们还探讨了一种新的变异策略，它确保变异的染色体仍然有效。不是直接修改染色体并加入随机性（在本例中，这可能导致无效的染色体），我们创建一个已知有效的随机 Individual，并用类似均匀交叉的方式，与它交换基因。该算法仍然被认为是均匀变异，但本章使用的新方法更容易确保有效的变异。

我们也结合了第 2 章的均匀交叉和与第 3 章的锦标赛选择，展现了遗传算法的许多方面是模块化的、独立的，能够以不同的方式进行组合。

最后，我们讨论了遗传算法的适应度得分的灵活性。我们了解到，无量纲的适应度得分可以用来引入软约束，并与硬约束调和，最终目标不只是得到有效的结果，而且是高品质的结果。

5.6 练习

1. 为排课程添加软约束。这可能包括教授偏好的时间和偏好的教室。
2. 实现支持一个配置文件或数据库连接，添加初始的时间表数据。
3. 编写一所学校的排课程序，要求学生每一个时段都有课。

第6章

优化

在本章中，我们将探讨优化遗传算法常用的各种技术。随着要解决的问题变得更加复杂，额外的优化技术就变得越来越重要。一个精心优化的算法可以节约数小时，在解决更大的问题时，甚至节约几天时间。因此，当问题达到一定复杂程度时，优化技术是必不可少的。

除了探索一些常用的优化技术，本章也将介绍一些实现的例子，它们用到前几章中案例研究的遗传算法。

6.1 自适应遗传算法

自适应遗传算法（Adaptive Genetic Algorithms，AGA）是遗传算法的一个流行子集，在合适的环境下使用时，可以提供超过标准实现的显著性能提升。正如我们在前几章中学到的，决定遗传算法性能的关键因素是它的参数配置方式。我们已经讨论过，在创建一个有效的遗传算法时，为变异率和交叉率找到合适的值的重要性。通常，配置参数需要一些试验和犯错，加上一些直觉，最终才能得到满意的配置。自适应遗传算法是有用的，因为它能帮助自动调整在

这些参数，基于算法的状态来调整它们。遗传算法运行时，会调整这些参数，希望得到的结果是，在执行过程中的任何特定时刻使用最佳的参数。正是算法参数的这种持续的自适应调整，常常导致遗传算法的性能改进。

自适应遗传算法使用一些信息，例如平均种群适应度和种群目前最佳适应度，来计算和更新它的参数，以便最适合其当前状态。例如，通过比较群体中任意特定个体和当前最适应的个体，有可能判断个体相对于当前最佳个体的表现。通常，我们希望增加保留表现良好的个体的机会，并降低保留表现不好个体的机会。一种方法就是，允许算法自适应地更新的变异率。

遗憾的是，没有那么简单。一段时间后，种群就开始收敛，个体开始靠近搜索空间中的单个点。发生这种情况时，搜索进程可受阻，因为个体之间的差别非常细微。在这种情况下，略微提高变异率，鼓励查找搜索空间的其他区域，这可能会有效。

通过计算当前最佳适应度和平均种群适应度之间的差，可以判断算法是否已开始收敛。如果种群平均适应度接近当前最佳适应度，我们就知道种群已开始收敛在搜索空间的一个小区域。

但是，自适应遗传算法不仅可以用于调整变异率。类似的技术可以用于调整遗传算法的其他参数，如交叉率，从而根据需要提供进一步的改进。

6.1.1 实现

正如与遗传算法相关的很多事情一样，更新参数的最佳方法通常需要一些试验。我们将探讨一种比较常见的方法，如果你愿意，可以自行尝试其他方法。

如前所述，计算特定个体应该采用什么变异率时，要考虑的最重要的两个方面是，当前个体的表现如何，以及整个种群作为一个整体的表现如何。用于评估这两方面并更新变异率的算法如下：

$$p_m = (f_{max} - f_i) / (f_{max} - f_{avg}) * m, f_i > f_{avg}$$

$$p_m = m, f_i \leqslant f_{avg}$$

如果个体的适应度高于种群的平均适应度，我们从种群中取得最佳适应度（f_{max}），并求出它与当前个体适应度（f_i）之间的差。然后，我们找到种群最佳适应度和种群平均适应度（f_{avg}）之间的差，并用这两个差相除。我们可以利用这个值来缩放初始化时设置的变异率。如果个体的适应度等于或小于种群的平均适应度，就使用初始化时设置的变异率。

方便起见，我们可以在前面的排课程序中，实现新的自适应遗传算法。首先，我们需要添加一个新的方法，获取种群的平均适应度。要做到这一点，可以在 Population 类的任何位置添加以下方法：

```
/**
     * Get average fitness
     *
     * @return The average individual fitness
     */
    public double getAvgFitness(){
        if (this.populationFitness == -1) {
            double totalFitness = 0;
            for (Individual individual : population) {
                totalFitness += individual.getFitness();
            }

            this.populationFitness = totalFitness;
        }
```

```
        return populationFitness / this.size();
    }
```

现在，可以利用我们的自适应变异算法更新变异函数，完成实现。

```
    /**
     * Apply mutation to population
     *
     * @param population
     * @param timetable
     * @return The mutated population
     */
    public Population mutatePopulation(Population population, Timetable
    timetable){
        // Initialize new population
        Population newPopulation = new Population(this.populationSize);

        // Get best fitness
        double bestFitness = population.getFittest(0).getFitness();

        // Loop over current population by fitness
        for (int populationIndex = 0; populationIndex < population.size();
        populationIndex++) {
            Individual individual = population.getFittest(populationIndex);

            // Create random individual to swap genes with
            Individual randomIndividual = new Individual(timetable);
            // Calculate adaptive mutation rate
```

```
        double adaptiveMutationRate = this.mutationRate;
        if (individual.getFitness() > population.getAvgFitness()) {
            double fitnessDelta1 = bestFitness - individual.
            getFitness();
            double fitnessDelta2 = bestFitness - population.
            getAvgFitness();
            adaptiveMutationRate = (fitnessDelta1 / fitnessDelta2) *
            this.mutationRate;
        }

        // Loop over individual's genes
        for (int geneIndex = 0; geneIndex < individual.
        getChromosomeLength(); geneIndex++) {
            // Skip mutation if this is an elite individual
            if (populationIndex > this.elitismCount) {
                // Does this gene need mutating?
                if (adaptiveMutationRate > Math.random()) {
                    // Swap for new gene
                    individual.setGene(geneIndex, randomIndividual.
                    getGene(geneIndex));
                }
            }
        }

        // Add individual to population
        newPopulation.setIndividual(populationIndex, individual);
    }
```

```
        // Return mutated population
        return newPopulation;
    }
```

除了实现上述算法的适应性变异代码之外，这个新的 mutatePopulation 方法与原来的等价。

在初始化带自适应变异的遗传算法时，现在所用的变异率是最大可能的变异率，并根据当前个体和种群整体的适应度进行缩减。因此，较高的初始变异率可能是有利的。

6.1.2 练习

利用你对自适应性变异率的理解，在遗传算法中实现自适应交叉率。

6.2 多次启发

在谈到优化遗传算法实现时，在某些条件下，二次启发是实现显著性能改进的另一种常用方法。在遗传算法中实现二次启发，让我们能将多种启发方法中最好的方面结合在一个算法中，对搜索策略和性能提供进一步的控制。

有两种流行的启发方法常常在遗传算法中实现，即模拟退火和禁忌搜索。“模拟退火”是仿照冶金退火过程的搜索启发方法。简单来说，它是一个爬山算法，旨在逐步减少较差解的接受率。在遗传算法的上下文中，模拟退火将随时间降低变异率和（或）交叉率。

另一方面，禁忌搜索是一种搜索算法，它保持一个“禁忌”（源自 taboo）解的列表，防止算法回到搜索空间中以前访问过的、已知是差的区域。这个禁忌列表有助于避免算法反复考虑它以前发现的、已知是差的解。

通常，多次启发方法，只有包含它能对搜索过程带来某种需要的改进时，才会被实现。例如，如果遗传算法在搜索空间的一个区域收敛太快，在算法中实现模拟退火，可能有助于控制算法收敛的速度。

6.2.1 实现

让我们来看多次启发算法的一个简单例子，它在遗传算法中结合了模拟退火算法。如前所述，模拟退火算法是一种爬山算法，它开始以较高的比率接受较差的解，然后随着算法的运行，它逐渐减小较差解的接受比率。

在遗传算法中实现该特性的最简单方法，就是更新变异率和交叉率，从较高的比率开始，然后随着算法的进行，逐渐降低变异率和交叉率。最初较高的变异率和交叉率，会导致遗传算法查找搜索空间的大块区域。然后随着变异率和交叉率慢慢降低，遗传算法应该开始专注于查找搜索空间中适应度值较高的区域。

为了改变变异和交叉概率，我们使用一个温度变量，它开始时高（或热），随着算法运行，慢慢地降低（或冷却）。这种加热和冷却技术直接源自冶金退火过程的启发。每一代的温度会稍微冷却，这降低了变异和交叉概率。

要开始实现，我们需要在 GeneticAlgorithm 类中创建两个新变量。coolingRate 应设置为一个很小的分数，通常量级是 0.001 或更少，尽管该数值将取决于你希望运行的世代数，以及模拟退火有多激进。

```
private double temperature = 1.0;
private double coolingRate;
```

接下来需要创建一个函数，基于 coolingRate 来冷却温度。

```
/**
     * Cool temperature
     */
    public void coolTemperature() {
        this.temperature *= (1 - this.coolingRate);
    }
```

现在，我们可以更新变异函数，在决定是否进行变异时，考虑 temperature 变量。要做到这一点，可以修改这行代码：

```
// Does this gene need mutation?
if (this.mutationRate > Math.random()) {
```

以使其包含一个新的 temprature 变量，如下所示：

```
// Does this gene need mutation?
if ((this.mutationRate * this.getTempature()) > Math.random()) {
```

要完成这个任务，更新执行类 main 方法中的遗传算法循环代码，在每一代结束时运行 coolTemperature( )函数。同样，你可能需要调整最初的变异率，因为它现在是基于温度值的最高比率。

6.2.2 练习

利用你对模拟退火启发方法的理解，将它应用于交叉率。

6.3 性能改进

除了改进搜索的启发方法，还有其他方法来优化遗传算法。可能优化遗传算法的最有效的方式之一，就是编写高效的代码。如果构建的遗传算法需要运行数千代，只要每代的处理时间减少几分之一秒，就可以大大减少总的运行时间。

6.3.1 适应度函数设计

因为适应度函数通常是遗传算法中计算量最大的部分，所以专注于适应度函数的代码改进，以期得到最好的性能回报，这是有意义的。

在对适应度函数进行改进之前，最好是先确保它充分地表示了问题。遗传算法利用适应度函数，衡量的搜索空间的最佳区域，在其中集中搜索。这意味着，设计不良的适应度函数会对搜索能力和遗传算法的整体性能产生巨大的负面影响。举个例子，假设一个遗传算法用于设计汽车面板，但这个评估汽车面板的适应度函数只考虑了汽车的最高时速。如果同样重要的因素还包括面板要满足一定的耐久性、要符合人体工程学的限制，以及符合空气动力学，那么这种过于简单的适应度函数可能无法提供足够的适应度值。

6.3.2 并行处理

现代计算机通常会配备几个独立的处理单元或“内核”。不同于标准的单核系统，多核系统都能利用其他核心同时处理多个计算。这意味着所有精心设计的应用程序，应能利用这一特性，让它的处理需求分布在可用的额外处理内

核上。对于一些应用程序，这可能就是在一个核上处理 GUI 相关的计算，并在另一个核上处理所有其他计算。

要在现代计算机上实现性能改进，支持多核系统的优势是一种简单而有效的方式。正如我们前面所讨论的，适应度函数往往是一个遗传算法的瓶颈。这使它成为多核优化的理想候选者。通过利用多核，它可以同时计算很多个体的适应度，如果每个种群通常有几百个个体要评估，这就大不一样了。

好在，Java 8 提供了一些非常有用的库，让我们在遗传算法中支持并行处理容易很多。利用 IntStream，我们可以实现适应度函数的并行处理，而无需担心并行处理的细节（如我们需要支持的内核数量），它根据可用核的数量，创建最佳数目的线程。

在第 5 章中，你可能想知道为什么 GeneticAlgorithm 的 calcFitness 方法在使用 Timetable 对象之前先克隆它。在并行处理的多线程应用程序中，需要注意确保在一个线程中的对象将不会影响在另一个线程的对象。在这个例子中，一个线程对时间表对象所作的改变，可能对相同时间使用相同对象的其他线程带来意外的结果，所以先克隆 Timetable 让我们为每个线程提供自己的对象。

我们可以修改 GeneticAlgorithm 的 evalPopulation 方法，利用 Java 的 IntStream，让第 5 章中的排课程序享受多线程的好处：

```
    /**
     * Evaluate population
     *
     * @param population
     * @param timetable
     */
    public void evalPopulation(Population population, Timetable timetable){
```

```
        IntStream.range(0, population.size()).parallel()
         .forEach(i -> this.calcFitness(population.getIndividual(i),
          timetable));

        double populationFitness = 0;

        // Loop over population evaluating individuals and suming
        population fitness
        for (Individual individual : population.getIndividuals()) {
            populationFitness += individual.getFitness();
        }

        population.setPopulationFitness(populationFitness);
    }
```

现在，如果系统支持，calcFitness 函数就能在多核上运行。

由于本书介绍的遗传算法使用了非常简单的适应度函数，并行处理可能无法提供太大的性能提升。有一个很好的方法来测试并行处理如何能提高遗传算法的性能，即在适应度函数中添加对 Thread.sleep()的调用。这将模拟需要相当多时间来执行的适应度函数。

6.3.3 适应度值散列

正如前面所讨论的，适应度函数通常是遗传算法中计算最昂贵的部分。因此，即使适应度函数的小改进，也可能对性能有很大影响。值散列是另一种方法，通过在散列表中保存以前计算的适应度值，能减少计算适应度值的时间。在大型分布式系统中，你可以使用一个集中式缓存服务（如 Redis 或 memcached），实现同样的目的。

在执行期间，由于随机变异和个体的重组，先前发现的解偶尔会被重新访问。随着遗传算法的收敛，并开始在搜索空间中越来越小的区域里寻找解，这种偶然重新访问的解就变得越来越常见。

每次重新访问一个解时，它的适应度值就需要重新计算，在重复、反复计算上浪费了处理能力。好在这很容易修正，只要在计算适应度值后，将它们保存在散列表中。在重新遇到以前计算过的解时，它的适应度值可直接从散列中取出，避免了重新计算。

为了在代码中添加适应度值散列，先在 GeneticAlgorithm 类中创建适应度散列表，

```
// Create fitness hashtable
    private Map<Individual, Double> fitnessHash = Collections.
    synchronizedMap(
            new LinkedHashMap<Individual, Double>() {
        @Override
        protected boolean removeEldestEntry(Entry<Individual, Double>
        eldest) {
            // Store a maximum of 1000 fitness values
            return this.size() > 1000;
        }
    });
```

在本例中，散列表最多保存 1000 个适应度值，超过就会移除最早的值。这可以根据需要修改，以便得到最佳的性能折衷。虽然较大的散列表可以容纳更多适应度值，但代价是占用更多内存。

现在，可以加入 get 和 put 方法，来检索和保存适应度值。将 calcFitness 方法更新如下，以实现这一点。请注意，我们已经移除了最后一部分的 IntStream 代码，以便能够每次对一个改进求值。

```
/**
     * Calculate individual's fitness value
     *
     * @param individual
     * @param timetable
     * @return fitness
     */
    public double calcFitness(Individual individual, Timetable timetable){
        Double storedFitness = this.fitnessHash.get(individual);
        if (storedFitness != null) {
            return storedFitness;
        }

        // Create new timetable object for thread
        Timetable threadTimetable = new Timetable(timetable);

        threadTimetable.createClasses(individual);

        // Calculate fitness
        int clashes = threadTimetable.calcClashes();
        double fitness = 1 / (double) (clashes + 1);

        individual.setFitness(fitness);

        // Store fitness in hashtable
        this.fitnessHash.put(individual, fitness);
```

```
        return fitness;
    }
```

最后，因为我们使用 Individual 对象作为散列表的键，所以需要重写 Individual 类的 equals 和 hashCode 方法。这是因为我们需要基于个体的染色体来散列，而不是对象本身，而默认是基于对象散列的。这很重要，因为具有相同染色体两个独立个体，应被适应度值散列表判定为相同。

```
    /**
     * Generates hash code based on individual's
     * chromosome
     *
     * @return Hash value
     */
    @Override
    public int hashCode() {
        int hash = Arrays.hashCode(this.chromosome);
        return hash;
    }

    /**
     * Equates based on individual's chromosome
     *
     * @return Equality boolean
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
```

```
        }
        if (getClass() != obj.getClass()) {
            return false;
        }

        Individual individual = (Individual) obj;
        return Arrays.equals(this.chromosome, individual.chromosome);
    }
```

6.3.4 编码

影响遗传算法性能的另一个方面是选择的编码方式。虽然在理论上，所有问题都可以用 0 和 1 的二进制编码来表示，但这很可能不是最有效的编码选择。

当一个遗传算法在为收敛而努力时，往往因为选择了不好的编码方式，导致在搜索新解时比较困难。没有硬性的规则来选择好的编码方式，但使用过于复杂的编码方式通常会产生不好的结果。例如，如果你希望编码方式能够编码 0～10 之间的 10 个数字，通常最好用 10 个整数的编码，而不是二进制字符串。这样它更容易应用变异和交叉函数，这些函数可应用于单个整数，而不是用位表示的整数值。这也意味着不必处理无效染色体，例如 1111 代表 15，超出了我们要求的范围 0～10。

6.3.5 变异和交叉方法

在考虑提高遗传算法性能的可选方案时，选好变异和交叉方法是另一个重要方面。最优的变异和交叉方法主要取决于选择的编码方式和问题本身的性质。好的变异或交叉的方法应该能够产生有效的解，还能够按照预期的方式对个体进行变异和交叉。

例如，如果我们要优化一个函数，它接受0～10之间的任意值，一个可能的变异方法是高斯变异（Gaussian mutation），它为基因增加了一个随机值，稍微增加或降低原来的值。然而，另一种可能的变异方法是边界变异（boundary mutation），它选择下限和上限之间的随机值，替换基因。这两种变异方法都能产生有效的变异，但根据问题的性质和实现的其他具体特点，一种可能优于另一种。不好的变异方法可能是简单根据原来的值，舍入为0或10。在这种情况下，发生变异的数量取决于基因的值，这可能导致较差的性能。初始值为1将被改变为0，这是相对小的变化。但是，初始值5将被改变为10，这种变化要大得多。这种偏差可能导致值倾向于接近0和10，这通常对遗传算法的搜索过程产生负面影响。

6.4 小结

遗传算法可以用不同的方式来修改，实现显著性能改进。在本章中，我们看到了许多不同的优化策略，以及如何在遗传算法中实现它们。

自适应遗传算法是一种优化策略，可以对标准遗传算法提供性能改进。自适应遗传算法让算法能够动态更新其参数，通常是修改变异率或交叉率。静态定义的参数不会根据算法的状态而调整，相比之下，这种动态的参数更新往往可以获得更好的结果。

本章考虑的另一种优化策略是多次启发。该策略包括遗传算法与另一种启发方法相结合，如模拟退火算法。通过结合另一种启发式的搜索特点，在这些特点有用的情况下，实现性能改进。本章中探讨的模拟退火算法，是基于冶金

的退火工艺。在遗传算法中实现时，它开始允许基因组中发生大的变化，然后逐渐减小变化的程度，让算法集中于搜索空间中有前途的区域。

要实现性能改善，最简单的方法之一就是优化适应度函数。适应度函数通常是最昂贵的计算部分，这使它成为理想的优化对象。适应度函数定义良好，并很好地反映个体的实际适应度，这也很重要。如果适应度函数不能很好地反映个体的表现，它会减慢搜索过程，将搜索导向搜索空间中较差的区域。

优化适应度函数有一种容易的方法，即支持并行处理。通过一次处理多个适应度函数，有可能大大降低遗传算法评估个体的时间。

要减少处理适应度函数所需的时间，另一个策略是适应度值散列。适应度值散列利用散列表，来保存最近用到的一些染色体的适应度值。如果这些染色体在算法中再次出现，就会回想起算过的适应度值，不用重新计算。这可以防止对求过值的个体重新求值。

最后，也可能有效的是考虑改进基因编码方法，或使用不同的变异或交叉方法，它们可以改善进化过程。例如，使用的编码方法不能很好地表示被编码的个体，或变异方法不能产生所需的基因差异化，这会导致算法停滞，从而导致产生较差的解。

Apress®

专业人士写给专业人士的书

Java遗传算法编程

遗传算法常常用于解决非常复杂的真实世界问题。

本书是学习如何利用遗传算法来解决问题的入门指南，书中包含了Java语言编写的、能运行的项目和解决方案。本书引导读者一步一步地实现各种遗传算法及一些常见应用场景，帮助读者在实践中加深理解，从而能够解决自己独特的问题。本书首先介绍了基本概念，并在随后的章节中添加了机器人控制、旅行商问题等例子，展示了实现遗传算法的更多知识技能。

通过阅读本书，你将熟悉遗传算法与编程语言相关的问题和概念，掌握构建自己的算法所需的全部知识，并且将获得用遗传算法解决问题的能力。请拿起本书，进入遗传算法这个迷人的领域，看看真正能工作的Java代码，并运用于你自己的项目和研究中。

本书具有以下特色：

- 引导学习遗传算法背后的理论；
- 解释软件开发者如何利用遗传算法来尝试解决一些问题；
- 通过简单易行的步骤，提供用Java实现遗传算法的详细指导。

异步社区
人民邮电出版社
www.epubit.com.cn

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

美术编辑：董志桢

分类建议：计算机 / 算法 / Java
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-43731-0

ISBN 978-7-115-43731-0
定价：49.00 元